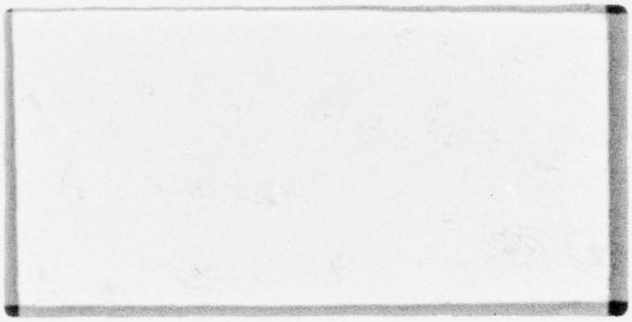
AD-A059 048 LOCKHEED MISSILES AND SPACE CO INC PALO ALTO CALIF PA--ETC F/G 12/1
SPARSE SYMMETRIC MATRIX PROCESSING.(U)
MAY 78 P S JENSEN, J K REID F49620-7 -0003
UNCLASSIFIED LMSC/D626184 AFOSR-TR-78-1149 NL

1 OF 1
AD A059048

LEVEL II

LOCKHEED
MISSILES & SPACE COMPANY, INC. • SUNNYVALE, CALIFORNIA
A SUBSIDIARY OF LOCKHEED AIRCRAFT CORPORATION



R FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
TICE OF TRANSMITTAL TO DDC
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is
approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

LEVEL II

(2)

(6) SPARSE SYMMETRIC MATRIX PROCESSING.

(10) Paul S. Jensen
John K. Reid

(9) Interim rept.

(11) 26 May 1978 (14) LMSC-D626184

(12) 96p.

(15) Report for AFOSR Contract F49620-76-C-0003

(16) 2304 (17) A3

(18) AFOSR (19) TR-78-1149

Research sponsored by the Air Force Office of Scientific Research (AFSC), under contract F49620-76-C-0003. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

LOCKHEED PALO ALTO RESEARCH LABORATORY
LOCKHEED MISSILES & SPACE COMPANY, INC.
A SUBSIDIARY OF LOCKHEED AIRCRAFT CORPORATION

JOB

TABLE OF CONTENTS

INTRODUCTION

This report includes three autonomous articles pertaining to the processing of large, symmetric, sparse matrices typified by those arising in the finite element analysis of boundary value problems. The work and computer programs discussed in this report resulted from a two-year effort sponsored by the Air Force Office of Scientific Research under contract F49620-76-C-0003.

The thrust of this part of the effort was to develop a sparse matrix processing system that could readily be incorporated in structural analysis computer programs to improve flexibility, computational efficiency and to reduce storage requirements. The requirements for the system were:

1. The ability to reliably process very large (out of core) problems, e.g., problems of order 15000;

2. The ability to utilize user knowledge about a given problem in order to reduce computational costs;

3. It must not depend upon user input for efficient operation and must be able to astutely "fill in" for incomplete user input;

4. The ability to exploit inherent computational efficiency resulting from repeated "substructures" appearing within a problem; and

5. It must be written in a readily transportable FORTRAN code for convenient adaptation to various computing machines.

A prototype sparse matrix processing system satisfying the above criteria was designed around the basic frontal technique augmented by a minimum-degree algorithm. The design resulted in a reasonably "natural" substructuring capability which was extended to satisfy criterion 4 above.

The prototype system was implemented and comparative tests were run using a widely accepted, general purpose structural analysis program as described in the first article of this report. The sparse matrix processing scheme used in that program is a very efficient implementation of the profile technique.

The second article in this report describes the new sparse matrix processing system SPSYST in detail. The version of SPSYST used in the first article differed slightly from the system described here in that a minimum fill algorithm was used in place of the minimum degree algorithm. The fundamental data management system used by SPSYST is based on a virtual memory approach. Since many computer systems do not provide virtual memory, a special FORTRAN virtual memory system was implemented as described in the third article of this report.

A COMPARISON OF TWO SPARSE MATRIX PROCESSING TECHNIQUES

P. S. Jensen

26 May 1978

ABSTRACT

The factor and solve capabilities of an established profile matrix processing program and a new sparse matrix program package are compared for several finite element analysis problems. The comparison tests were made on CDC CYBER 175 and UNIVAC 1110 computing systems. The problems run were analyses of reasonably simple shell structures. The new program was found to enjoy the greatest advantage for the more complicated problem geometries.

Section 1
INTRODUCTION

For this study we focus our attention on sparse, symmetric problems arising in structural analysis. Our objective is to gain a "feel" for the relative merits of two techniques for factoring very large sparse, symmetric matrices.

The profile technique, which evolved from band matrix techniques, is reasonably effective and has been the accepted "workhorse" for general structural analysis for many years. Basically, it is a band matrix technique that takes the semi-bandwidth of each row into account rather than assuming a constant bandwidth. A number of implementations of the profile technique are available including a convenient, unheralded version in a Lockheed library. It should be noted that the apparent simplicity of the profile technique is slightly deceptive and that subtle differences can have a significant effect on the overall efficiency. To quote N. Wirth, "the devil hides in detail".

General sparse matrix techniques [1, 3 and 4] are more recent and considerably less established. Potentially, they have an unquestionable advantage for very large problems because they involve fewer operations and deal with a smaller volume of data. In addition, they (potentially) are more flexible, e.g., they can factor certain perturbations of a given matrix without repeating the whole process and can utilize user knowledge of a problem to improve efficiency. The main weakness of the techniques is their relative complexity. The implementation tends to be a fairly large collection of complicated routines that require some skill on the part of a user for effective application. Although a number of implementations of sparse matrix techniques have been constructed over recent years, none appear to be suitable for very large problems that cannot be held in

the high speed memory of a computer. Consequently, in collaboration with Dr. John Reid, an implementation [2] that could be directly applied to important structural analysis problems was constructed for comparison with the profile technique. This implementation is briefly described in Section 2.

A widely used structural analysis program STAGS, briefly discussed in Section 3, was used to generate most of the test problems. It is implemented and normally run on CDC (Control Data Corporation) 6000 and CYBER series computers. Consequently, that computational environment was chosen for most of these tests. Specifically, the test runs were made on a CDC CYBER 175 computer using operating system SCOPE 3.4 and the standard "extended" FORTRAN 4. The architecture of the CDC computer tends to suit the profile technique somewhat better than the general sparse approach by virtue of its parallelism for arithmetic and slightly cumbersome logic capabilities. This natural bias arises from the facts that profile factorization is normally a Crout decomposition which involves a sequence of elementary vector operations (primarily dot product) on relatively long rows whereas the sparse factorization is a Gaussian elimination process involving relatively short "partial" rows.

Section 2
SPARSE MATRIX ALGORITHM

We are concerned with the direct solution of very large sets of linear equations

$$Ax = b \tag{2.1}$$

where symmetric, positive-definite matrix A can be expressed as a sum

$$A = \sum E^{(k)} \tag{2.2}$$

of matrices $E^{(k)}$ with non-zeros in only a relatively small number of rows and columns. Normally $E^{(k)}$ is a stiffness matrix associated with a single finite element; however, problems occurring in other applications can be handled provided only that each $E^{(k)}$ has relatively few non-zeros.

When we say that A is large we mean that we do not expect to be able to hold it in high-speed memory, even in packed form. Indeed we expect that sometimes one or more of the matrices $E^{(k)}$ may contain too many non-zeros to be packed into high-speed memory. We do however expect that sufficient high-speed memory is available for a number of variables that is a modest multiple of the order, n.

The method is a generalization of the frontal method [3]. Rather than directly reordering the variables of x, we instead reorder the sequence $\{E^{(k)}\}$ of (2.2) in order to reduce the total cost of factorization. Factorization is carried out simultaneously with assembly (summing of the $E^{(k)}$'s), completing the factorization on each row as soon as it is fully assembled. Note that if the results of the process through step $k > 1$ are saved, then it can be readily completed for a certain variety of matrices $E^{(k+1)}$ ...,

$E^{(m)}$ without ever repeating the first k steps. This property is important in the so-called "re-analysis" problem wherein a sequence of problems involving changes in a few element matrices $E^{(j)}$ must be solved.

The fundamental assemble/factor process is a pairwise operation, i.e., two matrices are added together and the "internal" variables (fully assembled rows) are eliminated before another pair of matrices are added together. The order in which the matrices are combined is thus conveniently described by a binary tree. On the basis of the topology (row/column position information for the non-zero data) of each $E^{(k)}$, a "good" tree structure can be constructed for a given set $\{E^{(k)}\}$ that prescribes a summation order requiring a minimal amount of computational cost. For economic reasons, one does not seek to develop the best possible tree but, instead, obtains one that works well and is not excessively expensive to construct. The present sparse matrix program uses a "min-fill" heuristic to construct the binary tree which, of course, needs to be exercised only once for a variety of actual nonzero values in the matrices $E^{(k)}$.

To further reduce the computational cost of constructing the binary tree, which we shall hereafter call the preprocessing, a system has been devised whereby the user can specify as much of the tree structure as he wishes. This is particularly useful for the analysis of physical problems for which certain natural groupings of element matrices $E^{(k)}$ are very evident to the analyst. For example, in the analysis of a propellor, it is natural to group the elements by blade for independent assembly. Such a group of elements is often called either a substructure or a super-element.

Summarizing some of the important benefits of the present sparse matrix processing system, we have:

- User knowledge of a problem can be readily utilized to reduce the solution cost;

- The user can arrange the computation to permit variation of the real data in a few $E^{(k)}$'s and obtain associated solutions at a minimal cost; and

- The process of substructuring is a natural function performed by this system. In fact, a means of setting values for substructure interface variables is provided so that solutions on any substructure can be obtained independently of other substructures.

Another convenient feature included in this sparse matrix factorization system is applicable when different subsets of the $E^{(k)}$'s in (2.2) are related in that they share the same basic topology but are shifted and permuted with diagonal symmetry in the assembled matrix. This situation would arise, for example, in modeling an assembly of just a few basic substructures as illustrated in Figure 2.1. In such a case, the factorization for each basic substructure needs to be carried out only once. It is then included in the analysis of the complete structure by a special same mechanism included in the computer program.

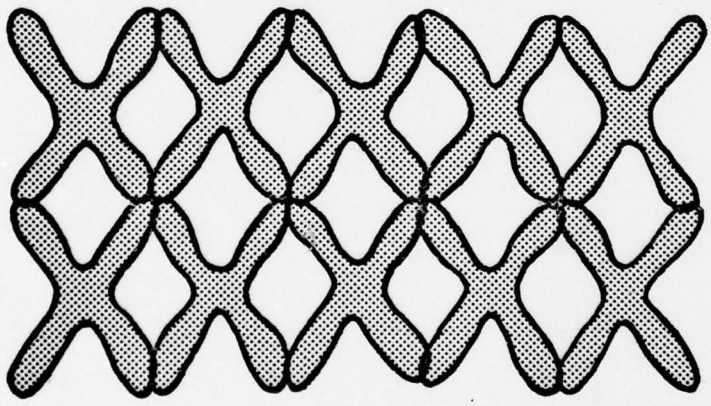


Figure 2.1. Illustration of a Simple Assembly

Data Management

A widely recognized "Achilles heel" of sparse matrix algorithms is the general overhead in dealing with the real data which are physically stored in esoteric ways to avoid the storage of zeros. This overhead is compounded when the data are so voluminous that they cannot be held in high-speed memory.

Referring to the entire collection of data for a given problem as the data space, we observe that at any point in time, the assemble/factor process tends to be focused only on a localized portion of the data space. This suggests that the adversity of a large data space can be relieved somewhat by a virtual memory system [6] that tends to hold actively used data in a high-speed memory buffer and move relatively inactive data to mass storage in a natural way.

Since many computers do not have virtual memory systems built in (hardware or software) a convenient, general purpose virtual memory system [7] was written (mostly in FORTRAN) for this purpose. The numerical experiments described in Sections 3 and 4 indicate that the virtual memory system is reasonably effective for the sparse matrix application.

Section 3
TESTS USING STAGS

The problems for this test were generated by the STAGS (Structural Analysis of General Shells) [5] program, which is widely used in the United States and Europe. The nonlinear and transient analysis of branched shells, which are common applications of STAGS, would appear to benefit considerably from the sparse matrix technique (Sec. 2) because of the many matrix factorizations required in each analysis and the fact that a branched shell is rather natural for substructuring.

Several cylindrical shell configurations were used for this test series. The basic "element" used for this STAGS discretization was the 32 freedom curved quadrilateral illustrated in Figure 3.1. It has seven displacement freedoms at each corner and one at the midpoint of each side.

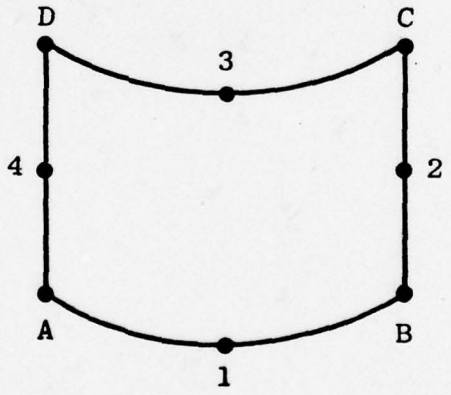


Figure 3.1. 32 Freedom STAGS Element. Seven freedoms at each of the corner nodes A,B,C,D and one freedom at each of the midpoint nodes 1,2,3,4.

All problems in this test series were run on a Control Data Corporation CYBER175 computer system. In both the sparse matrix and the profile algorithms, certain processes, e.g., vector inner product, vector copy and the general vector sum a $\underset{\sim}{x} + \underset{\sim}{y}$, were coded in assembly language. This fairly simple innovation tends to emphasize the adverse effects of the overhead costs inherent to the sparse matrix algorithms.

3.1 Problem Set 1

This set of problems involved a "square" cylindrical region having 3, 7 and 11 elements on each side. The resulting problem sizes were 137, 561 and 1273, i.e., the size is given by $9\,mn + 8(m + n + 1)$ where m and n are the number of elements in the two directions. Our objectives with this set of problems were to:

1. Assess the trade-off between large and small page sizes in the virtual storage system, keeping the high-speed buffer size fixed,

2. Compare the relative merits of the two subject algorithms on an indicative range of sizes of this simplest form of problem, and

3. Compare the automatic element ordering algorithm (minimal fill) with some astute orderings invoked by hand.

3.1.1 Page Size Trade-Off

The second problem (order 561) of this set was chosen to study the effects of varying the virtual system page size with a fixed buffer size. The factored sparse metrix for the problem had 22,975 non-zero elements and involved 710,241 multiply/add operations. Thus, the size of the problem was sufficiently large to give indicative results without being needlessly large. The input/ouptut activity and the central processor (CP) time

were studied during the factorization part of the processing for each of several page sizes and the results are displayed in Figure 3.2.

As one would expect, the number of physical I/O accesses increases and the I/O volume decreases with the number of pages in the fixed length buffer. The CPU time is relatively unchanged. Note from Figure 3.2 that the I/O volumes for 32 and 64 pages were approximately the same during factorization. This appears to be a result of the fact that the element matrix (size 528) was larger than the basic page size (448) in the 64-page case. Since the factorization generally is the major part of the work, it is generally wise to have the page size larger than the average element matrix.

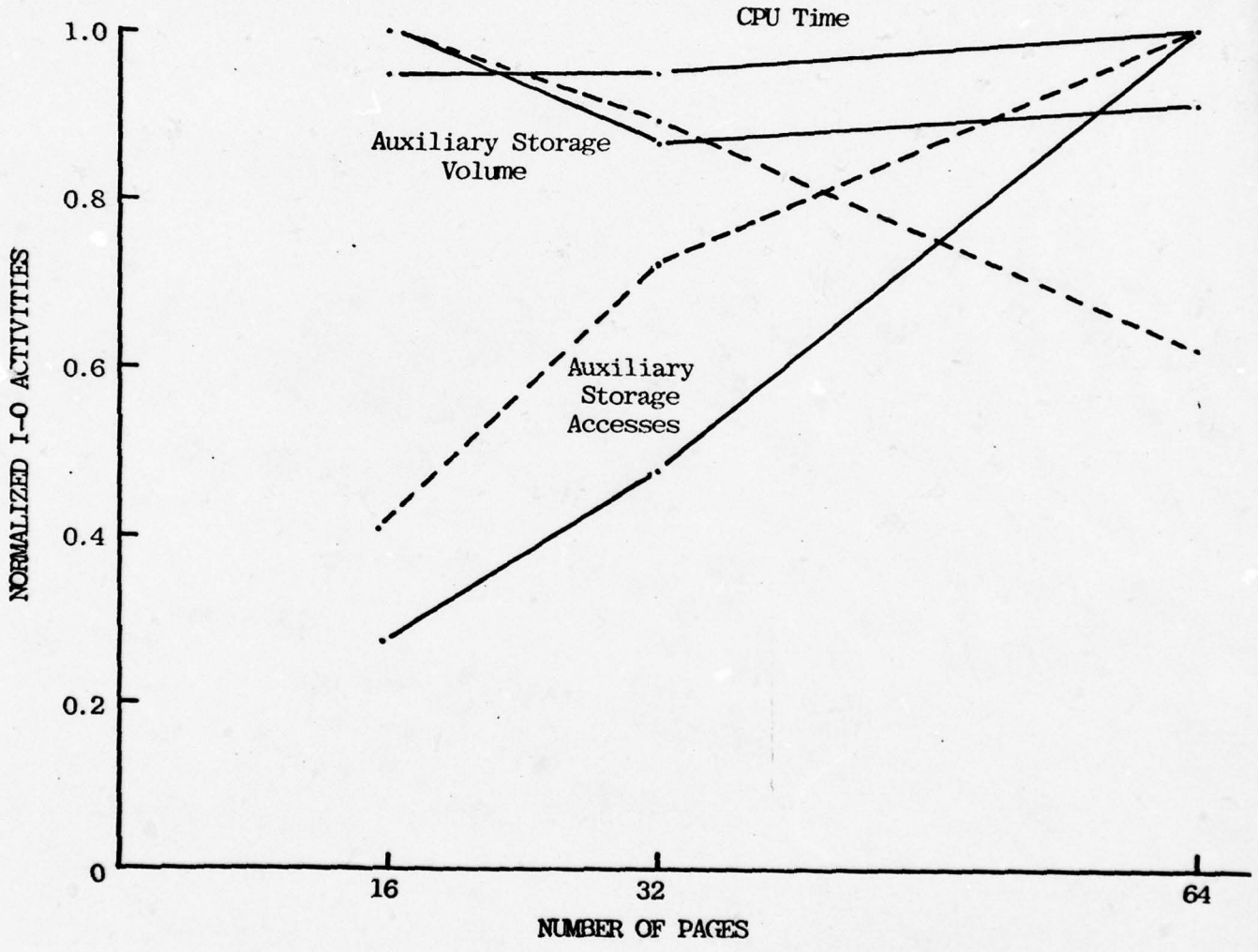


Figure 3.2. The effects on I/O processes during factorization resulting from varying the number of pages (and thus page size) in the high-speed memory buffer. Quantities are normalized by their maxima. Solid lines represent results obtained during factorization and dashed lines represent overall results. Buffer size = 28672.

As described in [7], a form of hashing algorithm is used to search the page rable of the virtual memory system. Sone interesting statistics about the search activity were recorded during these three tests and are summarized in Table 3.1.

Table 3.1

VIRTUAL SYSTEM OPERATIONAL STATISTICS FOR THREE TEST PROBLEMS

	Pages in Buffer		
	16	32	64
Ratio: Virtual Accesses to Physical Accesses	49	27	20
Ratio: Virtual Volume to Physical Volume	3.77	4.21	6.08
Average Table Search Steps per Page Find	1.34	1.29	1.33
Average Table Search Steps for No Find	0.62	0.54	0.60

3.1.2 Algorithm Comparisons for Simple Problems

In this test we sought to compare the "raw speed" of the sparse solver SPSYST and the highly tuned profile factorization system used in the STAGS structural analysis program. Since a number of efficiency improvements to SPSYST are planned whereas the profile system is about optimal, results favoring SPSYST here would be very encouraging.

The solution times for the three square panel test problems appear in Table 3.2. We observe that the profile factorization times were all better for these problems and that the relative advantage decreases with problem size. This trend substantiates the theoretical result which states that the computational work in the profile process grows at a higher rate (with respect to the problem size) than does the process in SPSYST. Thus, there exists a crossover point (greater than 1200) at which SPSYST would be faster than the profile method for this problem. Efficiency improvements in SPSYST will lower that crossover point; however, it is clear that the profile algorithm works very well for this simple problem class.

Table 3.2

COMPARATIVE RESULTS FOR SIMPLE, SQUARE PANEL PROBLEMS

Problem Size	Factor Time (sec)		Operations (in 1000's)		Non-Zeros (in 1000's)		Average Operation Time (micro-sec)	
	Profile	SPSYST	Profile	SPSYST	Profile	SPSYST	Profile	SPSYST
137	0.08	0.14	40.20	26.45	2.88	1.87	1.99	5.29
561	1.04	1.51	955.07	710.24	30.29	22.98	1.09	2.13
1237	4.48	6.00	5285.28	3582.60	110.75	80.72	0.85	1.67

3.1.3 Alternative Equation Ordering Systems

Aside from improving efficiency by changes to the implementation of SPSYST, we also wished to consider improvements by alternatives to the minimal fill algorithm used internally to order the variables. The basic substructuring capability of SPSYST provided a simple means for accomplishing this task, viz: the identification of certain groups of elements as substructures in the problem specifications.

An alternative ordering roughly equivalent to nested dissection, as illustrated in Figure 3.3, was used for this test. This certainly is not intended to represent a truly optimal ordering but simply to illustrate possible improvements.

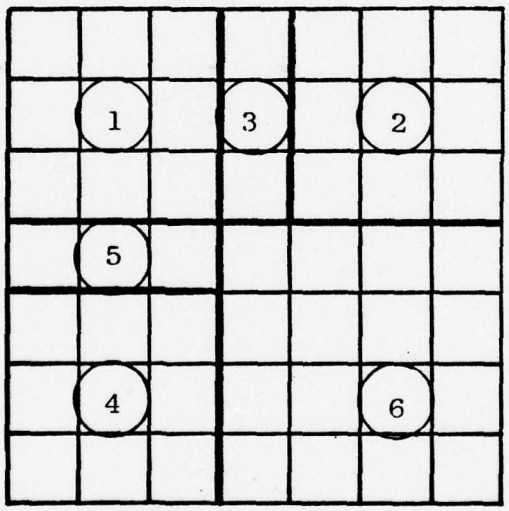


Figure 3.3. Substructuring used to obtain an alternative ordering for the 7x7 and 11x11 problems. The numbered regions were made super-elements in the order indicated.

Table 3.3

COMPARISON OF THE RESULTS USING THE
INTERNAL MINIMAL FILL ORDERING WITH THOSE
USING THE ALTERNATE ORDERING

	Problem					
	7 x 7			11 x 11		
	Min.Fill	Alternate	Ratio	Min.Fill	Alternate	Ratio
Factor Time	1.514	1.476	0.97	5.997	5.776	0.96
Solve Time	0.219	0.224	1.02	0.652	0.623	0.96
Nonzeros (in 1000's)	22.98	21.74	0.95	80.72	78.39	0.97
Operations (in 1000's)	687.71	595.88	0.87	3502.96	3206.95	0.92
μsec per Operation	2.20	2.48	1.13	1.71	1.80	1.05

Although the results in Table 3.3 indicate relatively small gains, they do indicate a potential for further work on the internal ordering algorithm

3.2 Problem Set 2

This set of problems consisted of branched shells, i.e., assemblies of smooth shell substructures (panels), where the joints may be "folds". This type of problem is much more typical of problems arising in practical engineering analysis than the simple square panels of the previous section. It is natural in problems of this nature to carry out the specification and analysis panel by panel followed by incorporation of the joint conditions for the complete structure.

This approach was applied to the "plus" shaped five branch problem illustrated in Figure 3.4. A small 20-element and a larger 57-element (longer arms) configurations were used with superior results from SPSYST in both cases.

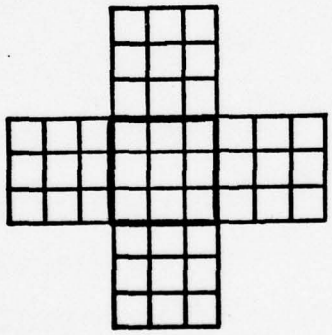


Figure 3.4. Plus-Shaped Problem

Normalizing the results with respect to the profile results, we obtained the relative statistics in Table 3.4 for SPSYST. Unfortunately, the operation times were so small that the accuracy of the times is questionable. This is evidenced by the rather erratic-looking results.

Table 3.4

RESULTS OF APPLYING SPSYST TO TWO PLUS-SHAPED REGIONS

(Cost factors are given relative to those for the profile solver)

	Case 1	Case 2
Elements	20.	57.
Problem size	308.	1009.
Nonzeros	0.40	0.47
Operations	0.16	0.19
Factor time	0.22	0.75
Solve time	0.08	0.34
Analysis time	0.50	1.90
Operation cost	5.12	3.95

It is evident that this slightly more complicated problem had a strongly adverse affect on the profile algorithm relative to the general sparse algorithm SPSYST and suggests that the advantages of SPSYST will be most noticeable for the more complex "real life" analysis problems. It is also evident that reducing the basic cost per operation in SPSYST is probably the most promising area of further research.

Section 4
TEST ON A UNIVAC 1110 COMPUTER

Because of the substantially different architecture of the UNIVAC 1110 computing system, it is interesting to obtain some comparative results on that system. Unfortunately, the same tests as in Section 3 could not be run on the 1110 because the STAGS analysis program was not operational on it. Consequently, the sparse program was exercised in conjunction with an analysis program called DIAL. The basic element used is illustrated in Figure 4.1. The profile factorization used by DIAL is an adaptation of that used in STAGS and so the comparison basis is reasonably close to that of Section 3.

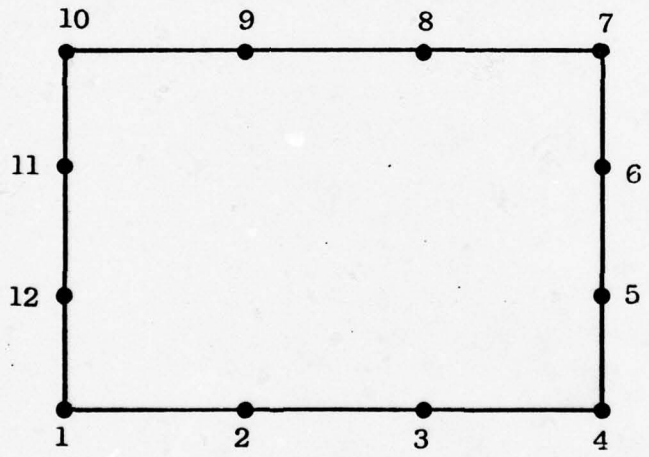


Figure 4.1. Basic DIAL 12-node element used for UNIVAC 1110 test. Two variables per node are defined.

A simple 10 by 10 square panel (100 elements) with 561 node points was analyzed, leading to a matrix equation of order 1119. The results obtained are given in Table 4.1. We notice that the ratio of 1.54 of the basic operation cost is substantially lower here than on CDC equipment where it ranged from 2-5 (see Tables 3.1 and 3.3).

Table 4.1

COMPARATIVE RESULTS FOR SOLVING
A 10x10 PROBLEM ON THE UNIVAC 1110 COMPUTER

	Profile	SPSYST	Ratio
Nonzeros (1000's)	95.12	56.94	0.60
Operations (1000's)	4200.00**	1723.06	0.41
Factor time	32.49	15.99	0.49
Solve time	1.92	1.75	0.91
Analysis time	14.59*	1.90	0.13
Operation cost (μsec)	6.02	9.28	1.54

* The analysis time for the profile technique includes the assembly. SPSYST carries out assembly and factorization simultaneously.

** Approximate operation count.

Section 5
SUMMARY

The series of comparative tests that have been made for this study are indicative but not conclusive of the relative merits of two sparse matrix processing schemes for problems arising in structural analysis. The tests in Section 3 were carried out on a CDC CYBER 175 computer which permits a limited amount of parallel computation. The results indicated slightly smaller computing costs for the profile technique on square domains and substantially larger costs for it on irregular (plus-shaped) domains. It is likely that a good bandwidth reduction algorithm could improve matters for the profile technique but we conclude that the sparse matrix processor SPSYST is a strong contender in this computing environment.

The test in Section 4 was carried out on a UNIVAC 1110 computer which has a better instruction set for logical operations and less facility for parallel computing. In this environment, SPSYST proved to be substantially superior even for a square domain.

The approximate cost per operation will always appear to be more for SPSYST than for the profile algorithm because of the greater complexity of SPSYST. However, it is likely that it can be improved somewhat by modifications of the implementation.

Of at least equal importance to the cost considerations alluded to above are considerations related to ease of use and flexibility. Presently, SPSYST is fairly difficult to use because of certain implementation details relating to data handling. With regard to flexibility, however, it is definitely superior to the profile algorithm. It permits substructuring in a natural way, efficiently handles the reanalysis situation wherein a relative few element properties are changed, and provides a powerful capacity for treating repeated substructures.

Acknowledgments

The author expresses his thanks for the well written and documented sparse matrix code provided by Dr. J. K. Reid which was the motivation for this test effort. Thanks also go to Mr. F. A. Brogan who provided the profile matrix processing code, an adaptation of the STAGS structural analysis program suitable for conducting the tests of Section 3 and substantial help in carrying out the tests. Finally, thanks go to Mr. G. Fergusson for adapting the sparse code for operation on the UNIVAC 1110 computer and running the test of Section 4.

REFERENCES

1. George, Alan, "A Survey of Sparse Matrix Methods in the Direct Solution of Finite Element Equations", Proc. 1973 Summer Comp. Councils, Inc., La Jolla, Calif. (July 17-19, 1973), pp. 15-20

2. Reid, John K., "A Package of Subroutines for Solution of Very Large Sets of Linear Finite-Element Equations", Article 2 of this report and report AERE-M.2947, Atomic Energy Research Establishment, Harwell, England (Feb. 1978)

3. Irons, B. M., "A Frontal Solution Program for Finite Element Analysis", Int. J. Numer. Meth. in Engr., 2 (1970), pp. 5-32

4. Eisenstat, S. C., M. H. Schultz, and A. H. Sherman, "Efficient Implementation of Sparse Symmetric Gaussian Elimination", in Advances in Computer Meth. for Partial Differential Equations", AICA (1975), pp. 33-39

5. Almroth, B. O., F. A. Brogan, and G. M. Stanley, "Structural Analysis of General Shells", Vol. 3 (User Instructions for STAGSC), Lockheed Missiles & Space Company, Inc. report, Sunnyvale, Calif. (Dec. 1975)

6. Denning, P. J., "Virtual Memory", (ACM) Computing Surveys, 2,3 (1970), pp. 153-189

7. Jensen, P. S., "A FORTRAN Virtual Storage Simulator for Non-Virtual Computers", article 3 of this report (March 1978)

A PACKAGE OF SUBROUTINES FOR SOLUTION OF VERY LARGE SETS OF LINEAR FINITE-ELEMENT EQUATIONS

J.K. Reid

February 1978 AERE-M.2947

Abstract

This report describes a package of subroutines designed to solve efficiently very large sets of linear finite-element equations whose matrix is symmetric and positive definite. It uses tree-search techniques to organise frontal elimination so that input-output operations are not excessive. It includes substructuring facilities and a good elimination order is found automatically.

CONTENTS

***

1. Introduction

We consider the efficient direct solution of very large sets of linear equations

$$Ax = b \tag{1.1}$$

whose matrix A is symmetric and positive-definite and can be expressed as a sum

$$A = \sum_k B^{(k)} \tag{1.2}$$

of matrices $B^{(k)}$ with non-zeros in only a relatively small number of rows and columns. Normally $B^{(k)}$ is a stiffness matrix associated with a single finite element, but problems occurring in other applications can be handled provided only that each $B^{(k)}$ has relatively few non-zeros (indeed we could express any matrix in this form by taking one $B^{(k)}$ for every non-zero).

We anticipate that the matrix A will often be so large that we are not able to hold it in main memory, even in packed form. Indeed we expect that sometimes one or more of the matrices $B^{(k)}$ may contain too many non-zeros to be held in packed form in main memory. However we do assume that the number of main storage locations available is a modest multiple of the order n. For instance we hold in main memory the right-hand side b, an integer work vector of length n and about 8 times as many integer pointers as there are matrices $B^{(k)}$.

It is our intention that our code be easy to incorporate in an existing program as a replacement for its present code for assembly (that is the summation (1.2)) and solution. To make the code portable we have used Standard Fortran, checked by the PFORT verifier (Ryder, 1974).

All input-output operations are performed through small auxiliary routines which request the reading or writing of an array of variables from or to a given file. Entries in the files are addressed exactly as if the file were a Fortran array. Indeed in the preliminary version described here the files actually are Fortran arrays. Our eventual intention is to replace them by a virtual memory system written in Fortran. Since most calls are soon followed by another call for adjacent entries in the same file, such a system will be efficient. For this reason the present code will run on a computer with a built-in virtual memory without excessive page thrashing.

To give a convenient and flexible interface to user programs we require the user to specify the problem by a series of subroutine calls. For instance each $B^{(k)}$ requires two subroutines to be called; the list of rows (and columns) containing non-zeros is specified by a call of INELV and the non-zeros themselves are specified by one or more calls of INELR. The order in which these subroutines are called is left almost entirely to the user's convenience, our only requirements being that the list of rows for a matrix $B^{(k)}$ must be specified before its real data and if several calls of INELR are made for one matrix $B^{(k)}$ then these calls must be consecutive.

The method used is a generalization of the frontal method (see for example Irons, 1970). Rather than seeking to order the variables for elimination we seek to order the assembly (1.2). Since addition and subtraction are associative operations, the elimination step

$$a_{ij}^{(\ell+1)} = a_{ij}^{(\ell)} - a_{i\ell}^{(\ell)} \, a_{\ell\ell}^{(\ell)-1} a_{\ell j}^{(\ell)} \qquad (1.3)$$

may be performed before all assembly steps

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} + b_{ij}^{(k)} \qquad (1.4)$$

are complete for a_{ij}, provided assembly is complete for the pivotal row (row ℓ). We actually perform the elimination operations (1.3) associated with the pivot $a_{\ell\ell}^{(\ell)}$ immediately after the ℓ^{th} row is fully assembled. It follows that the elimination order is determined from the order of assembly, apart from the ordering of rows that become fully assembled simultaneously. We represent the assembly order by a tree, an example of which is shown in Figure 1, and refers to the finite elements whose geometry is shown in Figures 2-5. The first assembly is of elements 1 and 2 and we call their union superelement 7. Similarly elements 3 and 4 are assembled into superelement 8. The geometry corresponding to this stage is shown in Figure 3. Next elements 5 and 6 are assembled into super-element 9, super-elements 7 and 8 are assembled into super-element 10 (see Figure 4) and finally 9 and 10 into 11 (see Figure 5).

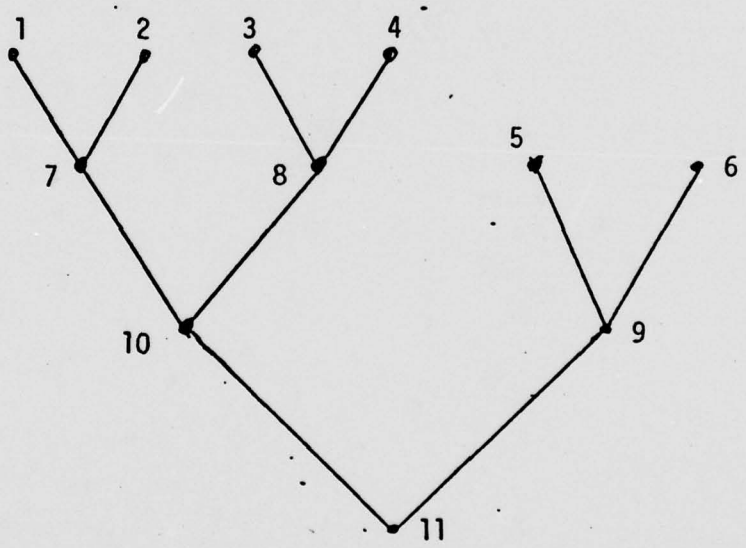


Figure 1 A simple assembly tree.

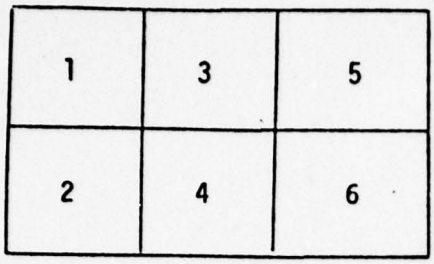


Figure 2 Original elements

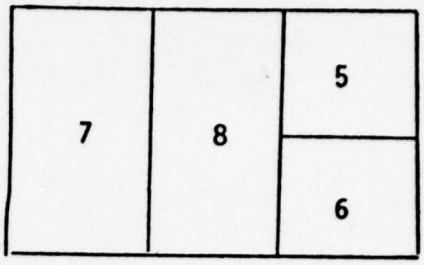


Figure 3 Superelements at level one of tree

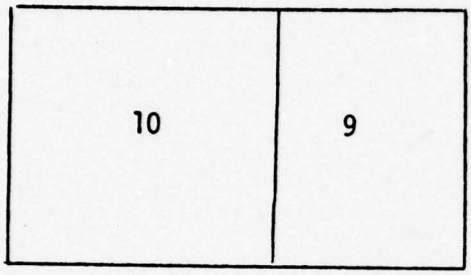


Figure 4 Superelements at level 2

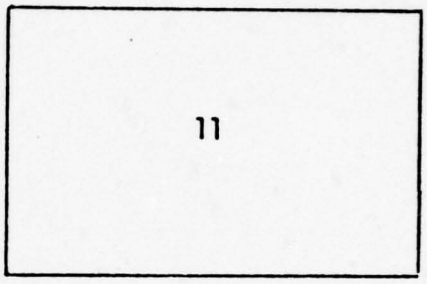


Figure 5 Final superelement

In referring to a tree such as that of Figure 1 we will say that node A is a"son" of node B and node B is the "father" of node A if node B is one level in the tree below node A and if the superelement corresponding to node B contains that corresponding to node A. Further we assume that different sons have different ages and will always draw them from left to right in order of decreasing age. For example node 10 in Figure 1 has elder son 7 and younger son 8. A node without a father (e.g. node 11 in Figure 1) is called a "root", and a node without a son (e.g. node 1 in Figure 1) is called a "terminal".

In our simple example we performed assemblies in the order 7 = {1,2}, 8 = {3,4}, 9 = {5,6}, 10 = {7,8}, 11 = {10,9}. However we could equally well have performed the assembly 10 = {7,8} immediately after 8 had been assembled. The arithmetic operations are exactly the same but there is an organisational advantage in that the need to store stiffness matrices 7,8 and 9 simultaneously is avoided and a stack can be used to hold intermediate results. Its contents will successively be the sets of matrices

$$\{7\} \begin{Bmatrix} 8 \\ 7 \end{Bmatrix} \{10\} \begin{Bmatrix} 9 \\ 10 \end{Bmatrix} \{11\} .$$

Such an order can be found in general by a depth-first search of the tree. In such a search we begin at the root and go from node to node as follows:

a) if the current node has a son not yet passed, go to the eldest such son

b) otherwise, if the node has a younger brother go to it

c) otherwise go to the father

until we return to the root. For Figure 1 the nodes would be passed in the order 11,10,7,1,2,7,8,3,4,8,10,9,5,6,9,11. Each time we return to a node we perform operations corresponding to assembling the stiffness matrices that correspond to its sons, performing all possible eliminations and storing the resulting matrix.

Since no node of the tree in Figure 1 has more than two sons (i.e. nodes linked to it from above) the way the assembly is to be performed is specified completely. The user is at liberty to provide such a tree, but we normally expect less detail than this and provide subroutines that fill in the detail (see section 6). For example the propellor shown in Figure 6 might have its elements grouped into blades and hub, as shown in the tree of Figure 7.

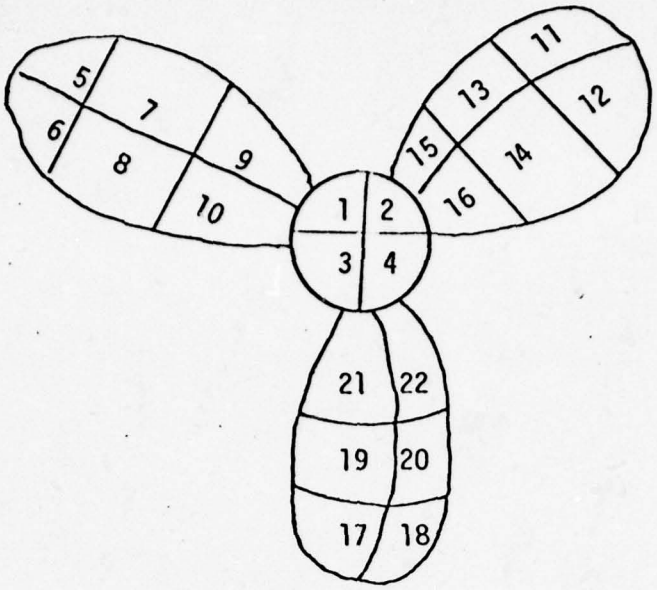


Figure 6 A propellor

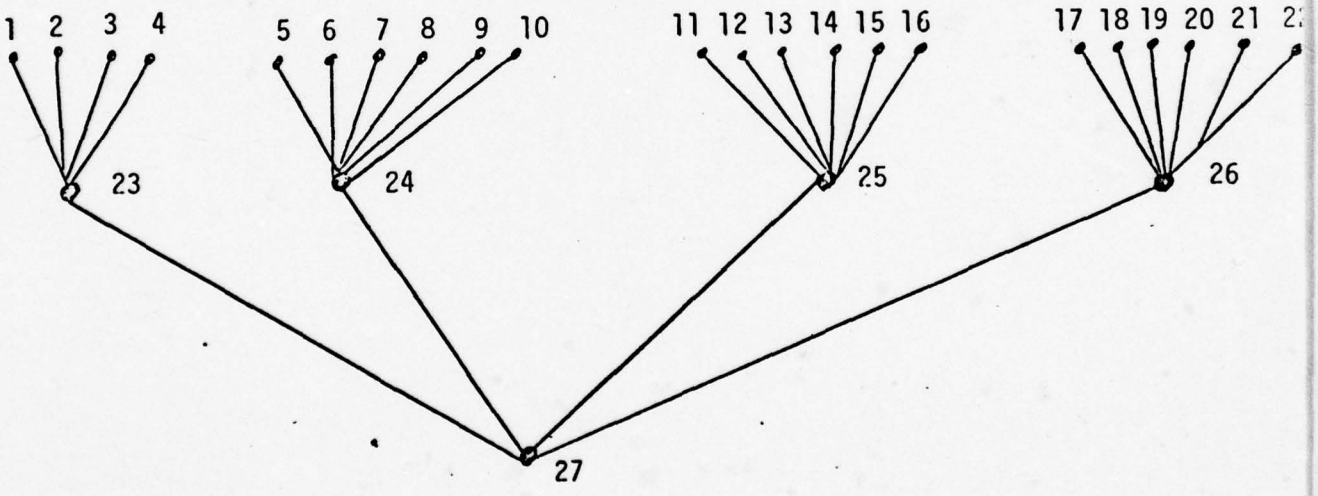


Figure 7 Tree for propellor

By the time a superelement is assembled, we can eliminate all variables internal to it, i.e. contained in no other elements. It is therefore advantageous to choose groupings with large numbers of internal variables. We believe that such groupings are often easily provided from the user's knowledge of the geometry of the structure.

Another feature of large structures is that they often contain repeating sub-structures. For example if the three blades in Figure 6 are identical then each (with suitable renumbering of the nodes) could be represented by the tree of Figure 1. There are obvious advantages in exploiting such features, and in these cases we join trees through "same interfaces" as illustrated for our propellor in Figure 8. Notice that the tree with root 32 represents any one of the blades.

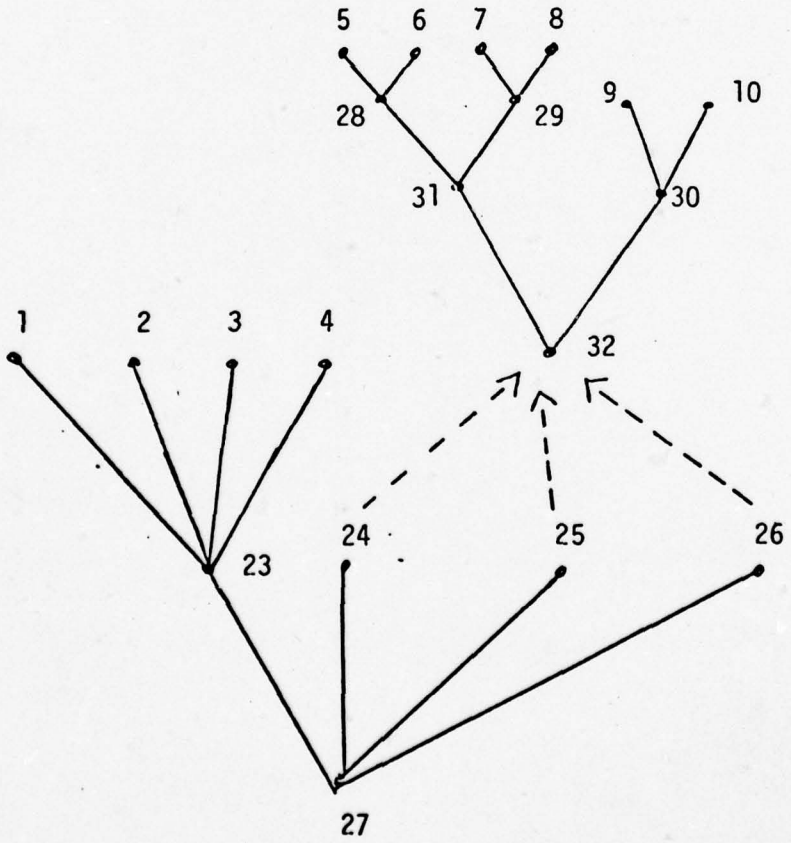


Figure 8 Trees and same interfaces for the propellor of Figure 6.

For each same interface the user must specify the correspondence between the variable names used in the single original (local names) and the variable names used in its copy in the overall structure (global names). For instance our propellor blade might have variable names as illustrated in Figure 9. In the overall structure blade 1 might have the same names and in blade 2 the names might correspond thus (2:22, 4:24, 5:21, 6:27, 7:28, 8:29, 9:32, 10:33, 11:30, 12:34).

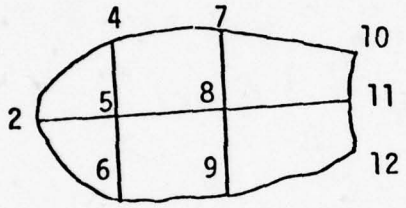


Figure 9 Variables of the original propellor blade

Besides the obvious storage advantage implied by the use of same interfaces, computing time may be saved by performing elimination operations once instead of several times. Similar computing advantages acrue if design changes are made in part of a structure, provided data associated with the old structure is preserved. For example if changes only in the hub of our propellor are made then everything associated with the tree representing the blades would still be valid and not need recomputing. Same interfaces provide natural points for the storage of partially factorized matrices and we give the user no other means of specifying such points. We therefore recommend the user to introduce extra same interfaces for substructures that may remain unchanged from one run to another, even though they may not be repeated in the overall structure.

We call each superelement that corresponds to the root of a tree a "root" superelement. Every same interface must point to a root superelement and the overall problem must be one too. With each we associate a (possible empty) set of variables called "boundary" variables. If these variables are permuted to the end then the overall matrix A and vector b take the form

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{1.5}$$

and the problem we actually solve is

$$(A_{11}\ A_{12}) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = b_1 \tag{1.6}$$

given values for the boundary variables x_2. In a heat-conduction problem, for example, the boundary variables might be measured temperatures on the boundary of the region. We do not require the user to make the permutation of equation (1.5), but do require him to specify which variables are boundary variables.

For other superelements a very similar role is played by any of its variables that are included in other elements and superelements in the overall structure and we call these boundary variables too. For the blade of Figure 9 the variables 10,11,12 are boundary, for example. In a root superelement that is used at same interfaces we judge whether a variable is boundary according to its disposition in the overall structure. We permit the solution of equations associated with any root superelement, interpreting the summation in equation (1.2) as being over those elements in the superelement. Thus equations associated with sub-structures may be solved separately given

values for the boundary variables.

The overall problem may also have an associated set of boundary variables, although we expect this to be unusual. The values of these variables must be specified for a solution to be obtained. It may be more convenient to do this than to eliminate them from the overall problem, but the factorization will be slower and require more storage because of the extra calculations associated with these variables. These calculations are, of course, necessary if it is planned to keep the factorization for later inclusion through one or more same interfaces in a bigger overall problem.

We use numerical names for the elements and variables. Element and superelement names must be distinct, although when same interfaces are in use the same element may of course appear several times in the overall structure. All the variables in the overall structure must have distinct names, but local names used within a root superelement across a same interface are regarded as private to that superelement and may be identical with names used globally.

2. Calls of input subroutines

To specify a problem, the user must first call INIT, an initializing subroutine and then make a series of calls to the four subroutines:

1) INELV to specify which variables are associated with a given element or are on the boundary of a root superelement.
2) INELR to specify the real data associated with an element (i.e. to specify one of the matrices $B^{(k)}$ of equation (1.2)).
3) INSUP to specify which elements and superelements belong to a given superelement.

4) INSAME to specify that a given superelement is identical with a given root superelement. The correspondence between the "local" names used for the variables in the original and the "global" names used in the overall problem must also be specified.

The order in which calls of the subroutines INELV,INELR,INSUP and INSAME are made is free to the user's discretion, except that a call of INELR may not be made before the call of INELV for the same element and if several calls of INELR are made to specify a single matrix $B^{(k)}$ then these must be consecutive.

All our subroutines contain an integer argument FLAG, which is used to indicate a prompt (positive values), a successful entry (zero value or an error condition (negative values). The full list of error conditions is given in section 5.

We now specify the action of the five input subroutines in detail.

2.1 Subroutine INIT

Subroutine INIT is called to specify the files to be used and to initialize data in main storage. A total of four files are used, two of which are for permanent storage of data about the problem and two for workspace. We have found it convenient to separate real and integer storage, so one of the permanent files is for reals and one for integers and similarly for the workspace files. It is our intention that these files eventually be out of main storage and then INIT will be used to specify their unit numbers. For the present COMMON blocks CRF,CIF,CRF2,CIF2 are used for the permanent reals and integers and the working reals and integers, respectively. Any four distinct integers are suitable for use when INIT is called.

The arguments and their purposes are as follows

SUBROUTINE INIT(MAXSUP,IIF,IRF,IRF2,LIIF,LIRF,LIIF2,LIRF2,FLAG).

Input-only variables

MAXSUP(INTEGER) must be set to the largest numerical name to be used for an element or superelement.

IIF,IRF(INTEGERs) must be set to the file numbers of the main files for integers and reals, respectively.

IIF2,IRF2(INTEGERs) must be set to the file numbers of the work files for integers and reals, respectively.

LIIF,LIRF,LIIF2,LIRF2(INTEGERs) must be set to the lengths (in Fortran storage units) of the four files (see sub-section 4.1 for some guidance on these lengths).

Output-only variable

FLAG(INTEGER) need not be set by the user. It is assigned the value 0 after a successful entry. If any of the other arguments is negative or if MAXSUP is too large, then the initialization is not performed and FLAG is set to -18.

2.2 Subroutine INELV

Subroutine INELV must be called for each element to specify which variables it involves. It may also be called for root superelements to specify which variables are "boundary variables" (i.e. are on the boundary of the overall problem or may be involved in other elements and superelements when the root superelement is included at a same interface in a larger superelement). The list may not contain any repeated variables. It is taken to be empty for any element or root superelement for which there has been no INELV call since the last call of INIT. It replaces any list previously input for the same element.

The arguments and their purposes are as follows:

SUBROUTINE INELV(NAME,NUM,LIST,FLAG)

Input-only arguments

NAME(INTEGER) must be set to the numerical name of the element or superelement.

NUM(INTEGER) must be set to the number of variables. If it is non-positive then the list is taken as empty.

LIST(INTEGER array of length NUM) must be set to contain the list of numerical variable names. Repetitions are not permitted.

Output-only argument

FLAG(INTEGER) need not be set by the user. It is set to zero after a successful entry. If the name of the element or any of the names of its variables is outside the permitted range, then a diagnostic is printed and FLAG is set to -10. If a variable is repeated then a diagnostic is printed and FLAG is set to -19. If the main integer file is too short then FLAG is set to -5. Under all these error conditions a message is printed and the list is ignored.

2.3 Subroutine INELR

Subroutine INELR must be called at least once for each element in order to specify the associated matrix $B^{(k)}$ (see equation (1.2)). It is assumed that $B^{(k)}$ has non-zeros only in those rows and columns that correspond to the variable list previously input for the element and the upper triangular part of the submatrix of these rows and columns is input by rows. If, for instance, the list is (3,7,1) then the non-zeros must be input in the order $b_{33}, b_{37}, b_{31}, b_{77}, b_{71}, b_{11}$. If the number of non-zeros is very large, then the list may be broken into parts and input by consecutive calls of INELR. The break points may be chosen to the user's convenience.

The arguments and their purposes are as follows:

SUBROUTINE INELR(NAME,ELEM,LEN,FLAG)

Input-only arguments

NAME(INTEGER) must be set to the numerical name of the element.

ELEM(REAL array of length LEN) must be set to contain the non-zeros being input.

LEN(INTEGER) must be set to contain the number of non-zeros being input. A non-positive value is taken to indicate an empty set of non-zeros.

Output-only argument

FLAG(INTEGER) need not be set by the user. It is set to zero after a successful entry, and to one of the values -7,-10,-14,-16 in the event of an error. Error conditions -7 (real file too small), -10 (name of element out of range) and -14 (data for previous element not complete) causes a diagnostic to be printed and the real data to be ignored while error condition -16 (too much data for element) causes a diagnostic message to be printed out but the data is stored.

2.4 Subroutine INSUP

Subroutine INSUP must be called for each superelement that the user wishes to specify. Calls may be made in any order. For instance if one superelement contains another it is not necessary for the inner superelement to have already been specified. Each call must specify a set of members to be added to a given superelement. Any that are present in another superelement are removed from that superelement. It is expected that normally all the members will be specified in one call but several calls may be made. Repeated names are permitted, later occurrences being ignored. The user may request that all elements and superelements not already included in a superelement be taken, and in this case he does not specify any list of names.

The arguments are as follows:

SUBROUTINE INSUP(NAME,NUM,LIST,FLAG)

Input-only arguments

NAME(INTEGER) must be set to the numerical name of the superelement whose members are being specified.

NUM(INTEGER) must be set to the number of members of NAME being specified, or to any non-positive value if all elements and superelements not already in a superelement are wanted.

LIST(INTEGER array of length NUM). If NUM≤0 then this array is not used, but otherwise it must be set to contain the numerical names of the elements and superelements that are to be members of superelement NAME.

Output-only argument

FLAG(INTEGER) need not be set by the user. It is set to zero after a successful entry. If the name of the superelement or any of its members is out of range then a diagnostic message is printed and FLAG is set to -10, the names in LIST that are within range being used even if one or more is out of range.

2.5 Subroutine INSAME

Subroutine INSAME is used to indicate that a given superelement is identical with a given root element or superelement. The correspondences between all the variable names used in the root-superelement (local names) and the names used for its copy in the larger superelement (global names) must be specified. Numerical names used as local names are regarded as private to the root superelement and may be reused as global names. The element and superelement names, however, are not reusable.

We refer to the correspondence established as a "same interface".

The arguments are as follows:

SUBROUTINE INSAME(NAME,NAMEOR,NUM,LIST,FLAG).

Input-only arguments

NAME(INTEGER) must be set to the numerical name of the superelement.

NAMEOR(INTEGER) must be set to the numerical name of the root superelement to which superelement NAME is identical.

NUM(INTEGER) must be set to the number of variables in the root superelement.

LIST(INTEGER array of dimensions (NUM,2)) must be set to contain the global variable names in LIST(I,1), I=1,2,...NUM and corresponding local names in LIST(I,2), I=1,2,...NUM.

Output-only argument

FLAG(INTEGER) need not be set by the user. It is set to zero after a successful entry. If NAME,NAMEOR or any of the variable names in LIST is out of range then a message is printed and FLAG is set to -10. If NAMEOR is not a root-superelement then a message is printed and FLAG is set to -12. If neither of these errors occurs the interface is stored and then a check is made as to whether the root superelement NAMEOR contains any variables not included in LIST(I,2),I=1,2,...NUM. If it does then a message is printed and FLAG is set to -13. If the check cannot be completed because array ELVAR is too small then a message is printed and FLAG is set to -4.

3. Calls of factorization and solution subroutines

Once the matrix A of equation (1.1) has been specified by a series of calls of the input subroutines a symmetric permutation of it may be factorized as

$$P^TAP = LU \qquad (2.1)$$

by a call of the subroutine FACTOR, provided it is associated with a root superelement. If the superelement has p boundary variables (see section 2.2) then U is upper triangular except in its last p rows and columns and L is unit lower triangular with elements:

$$\left.\begin{aligned} &\ell_{ij} = 0, && i<j \\ &\ell_{ii} = 1 \\ &\ell_{ij} = u_{ji}/u_{jj}, && i>j,\ j\le n-p \\ &\ell_{ij} = 0, && i>j,\ j>n-p \end{aligned}\right\} \qquad (2.2)$$

Once this factorization is established, the equation (1.1) may be solved by calling subroutine SOLVE which performs forward and backward substitution. If there are any boundary variables then SOLVE requests their values prior to back substitution.

Subroutine FACTOR begins by inserting extra superelements to ensure that no superelement has more than two component elements. This is done usi a minimal degree criterion, described in section 6. No real data is required for this analysis and indeed we recommend that no calls of INELR be made until after the first FACTOR call. It will then discover what extra superelements are needed and calculate the number of non-zeros in U and the number of real operations (multiplies and divides) necessary to find it. These quantities are stored in COMMON (see section 4). It then commences work on the real numbers, returning control to the user whenever it wants real data that is not available. The user is expected to make the relevant call of INELR and then recall FACTOR. One advantage of using this facility is that the real data will be stored in the order that it is required, whereas it is likely for essentially random access to be needed if it is all stored on file

beforehand. Another advantage is that an estimate of the work needed is provided early and might be used to terminate the run if the estimate is very high.

After FACTOR has been called, further calls of the input subroutines may be made to specify other parts of the overall problem or change existing parts. A subsequent call of FACTOR will not need to refactorize matrices associated with root superelements unchanged since previously factorized. Note that a root superelement is factorized both by a direct call of FACTOR and by virtue of being included via a same interface in a root superelement for which FACTOR is called.

We now specify the way subroutines FACTOR and SOLVE are called.

3.1 Subroutine FACTOR

Subroutine FACTOR factorizes the matrix associated with a root superelement and has the following arguments.

SUBROUTINE FACTOR(NAME,FLAG).

Input-only argument

NAME(INTEGER) must be set to the numerical name of the root superelement to be factorized.

Input-output argument

FLAG(INTEGER) must be set non-positive for initial entry. It has the value zero after a successful entry and a negative value in the range [-15,-1] in the event of an error (meanings are listed in section 5). A positive value of FLAG indicates a request to the user to load real data including that for element FLAG by one or more calls of INELR and then recall FACTOR with FLAG unchanged.

3.2 Subroutine SOLVE

Subroutine SOLVE solves the set of linear equations

$$Ax = b$$

associated with a root superelement. If the superelement has no boundary variables then a single call suffices but if there are any boundary variables then two calls must be made, one for the forward substitution operations and one for the back substitution with the correct values for boundary variables being set between the two calls. No change may have been made within the superelement since a call of FACTOR for it or for a superelement that contains it through a same interface.

The arguments are as follows:

SUBROUTINE SOLVE(NAME,MAXV,NSOL,B,FLAG).

Input-only arguments

NAME(INTEGER) must be set to the numerical name of the root superelement associated with A.

MAXV(INTEGER) must be set as least as large as the largest numerical name of a variable contained in superelement NAME.

NSOL(INTEGER) must be set to the number of columns in b.

Input-output arguments

B(REAL array of dimensions (MAXV,NSOL)). On initial entry (FLAG=0), B(i,j),j=1,2,...,NSOL must be set equal to the i^{th} row of b for each variable i contained in superelement NAME. On second entry (FLAG=1),B(i,j),j=1,2,...,NSOL must be set equal to the i^{th} row of x for each variable i on the boundary of superelement NAME and the rest of B must be left unchanged since return from the first entry. On final return (FLAG=0), B(i,j),j=1,2,...,NSOL will be equal to the i^{th} row of x for all variables in NAME. No other components of array B are altered.

FLAG(INTEGER) must be set to zero before initial entry and is zero after successful solution. It is set to 1 after forward substitution if NAME has any boundary variables and in this case the values of the boundary variables must be placed in B and SOLVE recalled with FLAG still equal to 1. Error conditions are indicated by the following values of FLAG

-10 NAME out of range.

-15 There is a zero pivot.

-17 MAXV or NSOL is non-positive.

-20 NAME has not been successfully factorized.

4. Storage

Information about the problem is held in two COMMON blocks called CRF and CIF which hold files of reals and integers, respectively and two COMMON blocks called CPOINT and SCALAR. This store is altered by the user calling the input subroutines (see section 2) to specify his problem or the subroutine FACTOR (see section 3) to factorize a matrix A associated with it and store the factors. A run may be terminated and restarted later provided that before termination these four COMMON blocks are filed and restored before restarting. In addition the COMMON blocks CSTACK,CELVAR, CA and CVAR are used for workspace and the COMMON blocks CRF2,CIF2 are used for workspace files. On most computer systems the user may increase the sizes of these arrays simply by declaring the required larger sizes in his program, but for conformance to standard Fortran he must make the same changes everywhere the arrays appear.

The purposes of the variables in COMMON block SCALAR (all INTEGERs) are as follows

LSTACK,LPOINT,...LVAR hold the last dimensions of the arrays STACK,...VAR.

IF holds the file number of the main integer file.

IFFR holds the position of the first free element in this file.

LIF holds the length in Fortran storage units of this file.

IF2,...LRF2 hold similar information for the integer work file, the main real file and the real work file, respectively.

INELRS holds the number of reals required to complete input of the last element matrix.

INELRN holds the name of the element whose real data was last input.

NZ is set by FACTOR to the number of reals it will add to the real file between original call (IFLAG=0) and final return (IFLAG=0). It is available on return with FLAG≥0.

NOP is set by FACTOR to the numbers of real multiply and divide operations FACTOR will perform (availability as for NZ).

MAXLEN is set by FACTOR to the largest number of variables in an element or on the boundary of a superelement (availability as for NZ).

MAXEL is the largest name so far used for a generated superelement.

NSTACK is used by FACTOR when returning with FLAG>0.

The array POINT, contained in COMMON block CPOINT, holds four integer pointers for each element and superelement. They are the following

POINT(1,IE): a) If positive: next younger brother in tree.
b) If negative: father in tree.
c) If zero: root of a tree.

POINT(2,IE): a) If positive: eldest son in tree.
b) If negative: pointer to original (same super-element).

POINT(3,IE): Pointer to integer file (value one for a null pointer).

POINT(4,IE): Pointer to real file (value one for null pointer). This is negated for elements not yet part of a factorized superelement.

Entries on the main integer file in COMMON block CIF are

a) for elements and root superelements

1. Number of variables in element or on boundary of superelement.
2. List of numerical names of these variables.
3. Pointers to the first and last entries on the real and integer files corresponding to eliminations associated with this root (superelements only). They are in the order first real first integer, last real, last integer.

b) for non-root superelements whose boundary list is shorter than the union of the boundary lists of its components.

1. Total number of variables.
2. List of numerical names of variables (internal names first).
3. Number of variables that are internal.

c) for same superelements

1. Number of variables in superelement.
2. List of global variable names.
3. List of corresponding local variable names.

Entries on the main real file in COMMON block CRF are:

a) for elements and root superelements with non-empty boundary: the upper triangular part of the stiffness matrix $B^{(k)}$ held by rows.

b) for non-root superelements whose boundary list is shorter than the union of the boundary lists of its two components: the rows of the factor U that correspond to the elimination of the internal variables.

Entries of type b) associated with a particular tree are held adjacently on both the real and integer file in the order corresponding to depth-first search of the tree, with one exception: whenever the search encounters a same interface pointing to another tree that may involve eliminations (i.e. one that is not a simple element) a single integer entry of -(root name) is inserted. All these entries may therefore be regarded as a single large elimination entry associated with the root, and entry a)3. for the root gives pointers to its beginning and end.

4.1 Sizes of arrays in COMMON

The package uses workspace in arrays in COMMON blocks and the purpose of this section is to explain briefly how each is used and what size is likely to be adequate. Throughout the package checks are made that array bounds are not exceeded and error messages are output and FLAG is set negative (see section 5) if any are exceeded.

The blocks with their default sizes are as follows

1)
```
COMMON/CSTACK/STACK(100)
INTEGER STACK
```

This array is used to hold a stack of node numbers beginning with the root and stretching through successive sons, sons of sons, etc. including cases where same interfaces are passed. Its length must therefore exceed the total number of tree levels. A (usually pessimistic) bound on the necessary size is the total number of elements, root superelements and same interfaces.

2)
```
COMMON/CA/A(10000)
REAL A
INTEGER ISUP, ISVAR,LLVAR
EQUIVALENCE (A(1),ISUP(1),ISVAR(1),LLVAR(1))
```

The array A is used as real workspace by subroutines FACTOR and SOLVE. A sufficient size is three times the greatest size of any

element or super-element (which size is recorded in MAXLEN of COMMON/SCALAR/), but for efficiency much greater sizes are recommended.

The equivalent integer array ISUP is used by ANAL1 to insert extra nodes in any part of the tree that consists of a father and several sons, and the way storage is used is explained in section 6.9. Storage may here be a problem if any node has an excessive number of sons. Further calls of INSUP to split the tree manually is an alternative to increasing the array size.

The equivalent integer arrays ISVAR and LLVAR are used by subroutines SFACT and SANAL (see sections 6.3 and 6.8) and must have length at least the greatest numerical name used for a variable.

3) COMMON/CPOINT/POINT(4,250)
INTEGER POINT

Array POINT holds four pointers for each tree node, i.e. for each element and super-element (see earlier in this section for details). Therefore its dimension must be at least as big as the biggest numerical name ever used for an element or super-element. The user specifies that his names lie in the range [1,MAXSUP] and extra names used for tree nodes created by the package begin at MAXSUP+1. The number created cannot exceed MAXSUP+(number of elements for which static condensation takes place). A safe size for the second dimension of POINT is therefore 3*MAXSUP.

4) COMMON/CELVAR/ELVAR(300)
INTEGER ELVAR

Array ELVAR is used for holding lists of variables associated with elements and super-elements and performing merges between them. A length of three times the longest such list (MAXLEN of COMMON/SCALAR/) is always sufficient. It is also used for a buffer

by SOLVE so there is some advantage in increasing its size beyond this lowest limit.

5) COMMON/CVAR/VAR(200)
INTEGER VAR

Array VAR is used to permit the rapid manipulation of lists of variable names. All its elements are set to zero by subroutine INIT and are normally reset to zero after use by any other subroutine. It is used, for example, by subroutine INELV to check for duplicate variable names. For each name M in the list, VAR(M) is checked for the value zero and is then reset to 1; if VAR(M) is non-zero when it is checked then variable M must have already occurred in the list. When all names have been checked the list is scanned again to reset the non-zeros values back to zero. The work performed is thus proportional to the length of the list.

VAR must have length at least the largest numerical name used for a variable. This is checked by subroutine INELV.

6) COMMON/CIF/IFL(5000)
COMMON/CRF/RFL(6000)
COMMON/CIF2/IFL2(1000)
COMMON/CRF2/RFL2(1000)
REAL RFL,RFL2
INTEGER IFL,IFL2

Arrays IFL,RFL,IFL2,RFL2 hold the main integer file, main real file, integer work-file and real work file, respectively. The amount actually used in each can be monitored by inspecting IFFR,IF2FR,RFFR,RF2FR in COMMON/SCALAR/ since they point to the first free location in each array. We recommend the user to adjust sizes in the light of his experience on his class of problems.

The largest array is almost always the main real file since this has to hold all the original element stiffness matrices and the factorized forms. The number of reals added to the file by FACTOR is given in NZ and is available on an intermediate return

(FLAG>0). The real work file can usually be significantly shorter, say by a factor of 5. The main integer file can also usually be shorter than the main real file. The ratio depends on the sizes of the original elements. With 60 variables in a typical element we have found it to be less than a tenth of the length, and with 12 variables in a typical element to be less than a quarter of the length. The integer work file is even smaller so is unlikely to cause problems.

5. Summary of error conditions

If any of the subroutines detects an error, then a diagnostic is printed and its argument FLAG is set to a negative value. The following is a full list of such error conditions.

-1 Array STACK is too small.

-2 Array A is too small.

-3 Array POINT is too small.

-4 Array ELVAR is too small.

-5 Main integer file is too small.

-6 Integer work-file is too small.

-7 Main real file is too small.

-8 Real work-file is too small.

-9 FACTOR called with FLAG out of range.

-10 A name is out of range.

-11 Real data not given to FACTOR when requested.

-12 Name used in call of FACTOR or for original of same set of superelements is not a root superelement.

-13 Faulty same interface (one or more variables missing).

-14 Real data is incomplete for an element.

-15 A zero pivot has been found.

-16 Too much real data supplied for an element.

-17 MAXV or NSOL non-positive on call of SOLVE.

-18 Faulty call of subroutine INIT.

-19 Variable repeated on call of INELV.

-20 Call of SOLVE for superelement that has not had a successful call of FACTOR.

6. Descriptions of individual subroutines

Our purpose in this section is to give brief descriptions of the way the subroutines work. The initialization subroutine INIT and the input subroutines INELV,INELR,INSUP and INSAME are straightforward and need no description beyond that already given in section 2. The task performed by FACTOR, however, is complicated and we have found it convenient to subdivide it into a total of eight subroutines. Separate descriptions of these are given in sub-sections 6.2 to 6.9, following a description in section 6.1 of how the many depth-first searches used by these subroutines are coded in Fortran. SOLVE and the small subroutine it calls also deserve some description and we do this in sub-sections 6.10 and 6.11. Almost all subroutines include calls to subroutines IOGETI,IOGETR,IOPUTI,IOPUTR to get from or put on file integer or real data, and these are described in sub-section 6.12.

6.1 Depth-first tree searches

Most of our subroutines perform one or more depth-first tree searches and the purpose of this sub-section is to explain how this is done.

We illustrate with the tree shown in Figure 10, which is that of Figure 8 with additional nodes inserted so that each root node has one son and no node has more than two sons. Such additional nodes are automatically inserted by ANAL1, called through FACTOR. The tree is stored in POINT(1,·) and POINT(2,·), as detailed in section 4. The values of these pointers for part of the tree of Figure 10 are shown for illustration in Table 1.

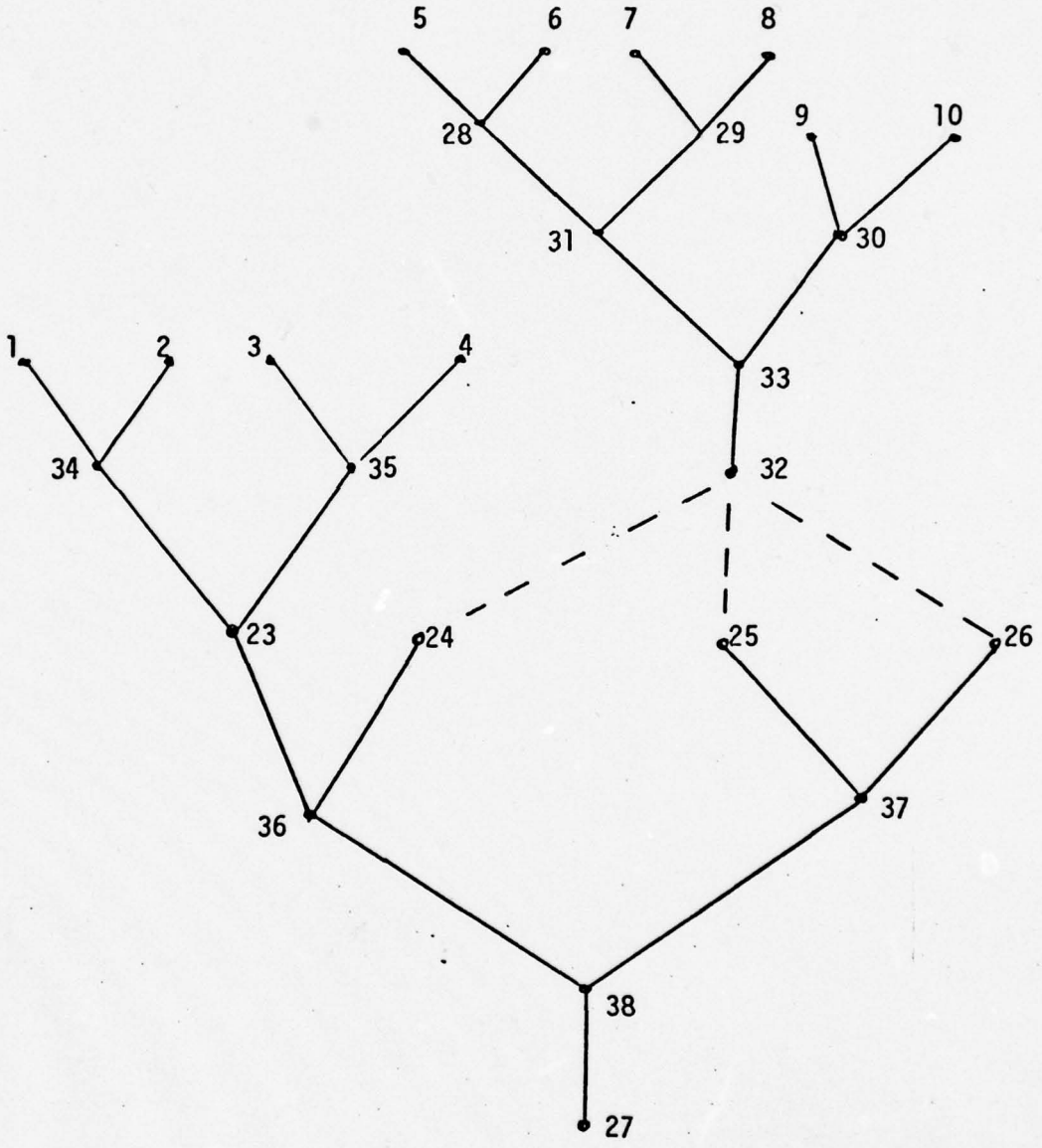


Figure 10 Tree for propellor with some additional super-elements.

IE	POINT(1,IE)	POINT(2,IE)
1	2	0
2	-34	0
3	4	0
4	-35	0
23	24	34
24	-36	-32
25	26	-32
26	-37	-32
28	29	5
29	-31	7

Table 1 Some of the pointers corresponding to Figure 10

To perform a depth-first tree search without crossing same interfaces (which for the example of Figure 10 would involve nodes 27,38,36,23,34,1,2,34,35,3,4,35,23,24,36,37,25,26,37,38,27) involves code of the form

```
      IE=NAME
      DO 70 NSCH=1,LPOINT
      I=POINT(2,IE)
      IF(I)20,30,10
C
C     IE HAS I AS SON
10    IE=I
      GO TO 70
C
C     IE IS A SAME INTERFACE POINTING TO ROOT SUPERELEMENT -I
20    GO TO 40
C
C     IE IS AN ELEMENT
30    GO TO 40
C
C     BACKTRACK UNTIL A NODE WITH A YOUNGER
C     BROTHER IS FOUND
40    DO 60 J=1,LPOINT
      IE=POINT(1,IE)
      IF(IE)50,80,70
50    IE= -IE
60    CONTINUE
C
70    CONTINUE
80
```

DO loops are used so that the loop structure is apparent. It is not expected that they will ever reach termination since LPOINT is the maximum number of tree nodes. The main loop DO 70 is executed once for each tree node. If the node has a son then it is replaced by the son. Otherwise the loop DO 60 is used to backtrack through successive fathers until a node with a younger brother or the root is reached.

Code of this kind appears in many places. The exact action taken as each node is passed varies from place to place but the overall structure remains.

Some modification is needed if we wish to cross same interfaces because we do not store at root super-elements any list of same interfaces in which they are involved. In this case we use the array STACK to hold a stack of integers consisting of the current element, its father, its father's father,..., the root. We regard the father-son relation as (temporarily) holding across any currently active same interfaces. The backtrack step then consists of popping the stack rather than using POINT(1,·).

6.2 Subroutine FACTOR

Subroutine FACTOR performs two depth-first tree searches. The first checks whether any node has more than two sons, whether any root node has more than one son or has a son that is an element or same interface. Any of these conditions indicate that additional nodes need to be inserted. The actual insertion is done by subroutine ANAL1, which is called through subroutine ANAL. Therefore a flag (IANAL) is set during the depth-first search if any such additional nodes are needed and ANAL is called at the end of the search if this flag is set.

This depth-first search also sets positive the signs of POINT(3,·) and POINT(4,·), which are used as flags at various places. An important example is that a negative value is placed in POINT(4,NAME) during any call of INELV,INELR,INSUP or INSAME for node NAME to indicate that information stored in association with it has (probably) changed. Any root super-element whose tree contains such a node (even indirectly through same interfaces) will require associated matrix factorization operations to be performed. Therefore during this first depth-first search any node with a son having a negative value for POINT(4,·) has its real file pointer replaced by the dummy value -1. Since this pointer is itself held in POINT(4,·), the negative value will result in the same action occurring for all its ancestors. In particular this will happen for all root super-elements containing the change and hence we can recognise where refactorization is needed.

On completion of this search, if POINT(4,·) is positive for the root of the tree being treated, then no change can have taken place since the last call of FACTOR and so an immediate RETURN is possible. Otherwise a call of ANAL is made if necessary and then a second depth-first search is executed. Each same interface encountered is checked by calling CKSAME (described in sub-section 6.6) and the interface is crossed only if the value of POINT(4,·) for its root indicates that it needs treatment. On returning to a same interface that has been crossed or on returning to the original root super-element the subroutine SFACT is called. This performs the actual factorization of the matrix associated with a root super-element on the assumption that the matrices associated with root super-elements contained in it through same interfaces have been factorized. Calling SFACT during a depth-first search ensures that this condition holds. SFACT itself is described in sub-section 6.3.

If called for root node IR, it leaves a positive value in POINT(4,IR) to act as a flag to prevent refactorization if the same root super-element is encountered later in the depth-first search.

6.3 Subroutine SFACT

Subroutine SFACT is called by FACTOR and factorizes the matrix associated with a root super-element on the assumption that all root super-elements contained in it through same interfaces have been factorized. If such an embedded root super-element has m boundary variables then a stiffness matrix having m rows and columns is associated with it and can be treated as if it were a stiffness matrix associated with an original element (apart from the change from local to global variable names). It follows that depth-first searches are possible using the tree pointers stored in array POINT, as in the code of sub-section 6.1, and that the code is little more complicated than it would be for a super-element containing simple component elements.

The first task of SFACT is to identify the pivotal sequence. The order in which elements (or super-elements) are amalgamated into larger super-elements is determined by a depth-first tree search during which an amalgamation is performed every time a father is reached from his youngest son. These amalgamations are first performed symbolically using index lists only. A stack is used to hold those lists that have been read or constructed but are not required immediately. As each terminal node is reached the index list of the corresponding element is loaded onto the top of the stack. At each back-tracking step to a father from his youngest son, the top two lists on the stack correspond to his two sons. These lists are removed from the stack, then merged and components corresponding to variables internal to the resulting super-element are removed. The resulting list is placed on the top of the stack. A variable

can be identified as internal by the fact that it does not appear elsewhere in the stack nor in the lists of elements reached later in the search nor in the boundary list of the root. A preliminary tree search is therefore necessary to find the last occurrence of every variable and it is convenient to regard the boundary list of the root as the last list encountered. It is also necessary to have a pointer for each variable to its first occurrence on the stack. As each variable is found to be internal to a super-element, it is placed next in the pivotal sequence so that the complete sequence is known at the end of the search.

An array (ELVAR) in main storage is used to perform the merging of index lists, and it is convenient at times to regard it as the top member of the stack. Each element index list is first read into ELVAR and is transferred to the stack itself only if another stack entry on top of it is anticipated. Similarly once a merged list has been constructed in ELVAR and variables corresponding to internal variables have been removed, it is left there until a stack entry on top of it is expected. If, on the other hand, the next operation is another merge then the top entry of the actual stack (which is really the second stack entry) is also loaded into ELVAR ready for the merge. For the actual stack we use the integer working file.

On the completion of these operations the position in the pivotal sequence of any variable M involved will be stored in VAR(M) and the number of eliminations associated with tree node IE will be stored (temporarily) in POINT(3,IE).

The actual elimination is performed during a final tree search. A stack of reals is held in the real work file. As each terminal node is reached the associated stiffness matrix is permuted to pivotal order and loaded onto the top of the stack by subroutine LOAD (see sub-section 6.4) and

as each father node is reached the associated assembly and elimination operations are performed by subroutine ELIM (see sub-section 6.5). Also whenever a terminal node that is a same interface is reached then an entry on the main integer file is made consisting of the single integer which is the negation of the name of the root super-element involved, unless this is a simple element. This permits the forward and back substitution operations of subroutine SOLVE to be performed under the control of file entries without a tree search.

6.4 Subroutine LOAD

Subroutine LOAD is called by subroutine FACTOR to read an element stiffness matrix, permute it to correspond with the pivotal order and place it on top of the stack.. The stack itself is held in the form of integer entries on the integer work file and real entries on the real work file.

The first task of LOAD is to read the length, LEN, of the list of variables associated with the element under consideration, then read the list itself, permute it to pivotal order and output the permuted list to the integer work file. To permit successive entries to be later read from the top of the stack without storing additional pointers, the value of LEN itself is written immediately after the permuted list. During the sort the original positions of the variables in the permuted list are stored, for use when permuting the reals.

The reals must now be permuted without necessarily there being enough workspace to hold them all. The upper triangular part of the matrix is held by rows on file and to find all elements of a particular row of the matrix, say row i, requires a scan of the i^{th} column of the upper triangle as well as the i^{th} row (see Figure 11). We have chosen to keep LEN locations in

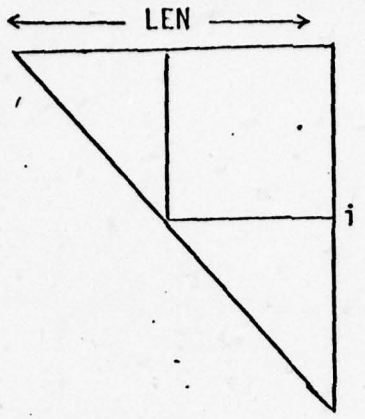


Figure 11 Scanning the elements of a whole row

work-array A free for creating the permuted rows and another LEN locations free for reading the corresponding unpermuted row, while using the rest to hold as much as possible of the unpermuted matrix. We read each row explicitly from file only if it is not already stored in A. For each element of the permuted row we determine whether it is held in A and read it explicitly from file if it is not. The rows are written out one-by-one. It can be seen that the efficiency depends on ample space being available in A, but execution can continue provided it has length at least 2*LEN.

6.5 Subroutine ELIM

Subroutine ELIM is called to perform the assembly and elimination operations associated with a node of the tree. The node may have a single son, in which case no assembly is necessary, but eliminations (corresponding to static condensation) are probably required. Normally, however, the node has two sons and assembly of the corresponding matrices is required.

The index list of the elder (or only) son is first read. In the single son case no merging of integers or of reals is necessary, but to give consistency with the result in the two-son case the reals are copied across from the work file (which holds the stack) to the main real file.

In the two-son case the second index list is read and is merged with the first. During this merge the original index lists are overwritten by the positions of the corresponding variables in the merged list, since these are required for the merging of the reals. Note that both index lists will be in pivotal order since this is how they are always left by LOAD and ELIM, so a single merge (without a sort) is adequate.

For the actual merge the array A is divided into three parts of equal length. The first part is used to accumulate rows of the merged matrix and the others hold rows of the original matrix. The merging is performed row by row. If there is not enough room for the whole of a row of the merged matrix, then those rows that have been constructed are output. Similarly if the whole of the corresponding row of either of the original matrices is not available, then as much as possible of the remaining matrix is read in. The resulting merged matrix is written to the real file. Notice that in a large case the records read and written will be long and, apart from a little overlap between reads, there is no unnecessary reading or writing.

The actual elimination operations are performed in several passes during each of which as many eliminations as the size of array A permits are executed. Each begins by copying the matrix from the main file to the stack. During the sequence of eliminations data is read from the stack and the modified data is written to the main file. The pass begins by reading as much of the matrix as possible into array A, and each row in turn has all relevant elimination operations applied to it. If all the rows

to be used as pivotal are available at this time, and there is enough additional space for at least one more row, then the pivotal rows are written out once they have been calculated but are held at the front of the array. The rest is used as a buffer for processing the remaining rows. If however not all these pivotal rows can be held at once then only as many pivotal operations as the above procedure permits are performed. Subsequent operations are performed on later passes. The final pass merely copies the uneliminated matrix across to the work file.

6.6 Subroutine CKSAME

Subroutine CKSAME is called by INSAME and FACTOR to check the lists of variables associated with a same interface and if necessary reorder them so that the boundary variables head the list and are in the correct order. It begins by performing a depth-first search of the tree whose root is at the interface to see if it involves just those variables that are in the associated list. It is assumed that terminal nodes of this tree that are themselves same interfaces have correct lists of variables. It then checks that the boundary variables are in their correct positions. If they are not then it sorts them and writes the sorted lists out to the main integer file.

6.7 Subroutine ANAL

Subroutine ANAL is called by FACTOR if it has found that extra nodes need to be inserted anywhere in the trees involved. FACTOR sets the signs of POINT(3,·) and POINT(4,·) positive but does not indicate which tree requires further nodes because it is not expensive to rediscover this. Any root super-element with an associated stiffness matrix certainly does not require such extra nodes, so we begin with a depth-first search in which all other root super-elements are marked for checking by setting POINT(3,·) negative. As each is checked this flag is reset positive so that it is checked only once.

A second depth-first search is performed to call subroutine SANAL for each such root super-element.

6.8 Subroutine SANAL

Subroutine SANAL is called for each root super-element whose tree may require extra nodes. It does not cross any same interfaces.

It begins with a depth-first search during which the last occurrence of every variable is recorded in array VAR. A second depth-first establishes lists of variables active at each node. Whenever a father node is reached from its youngest son the index lists of all its sons will have been established. Its own list is constructed by merging these lists and removing all variables internal to the corresponding super-element. Such variables must be involved only in elements associated with nodes which are descendants. These can be recognised by keeping a record of the lowest tree level at which each variable has so far appeared (in LLVAR) as well as the record of its last occurrence in array VAR. If the current node is at a lower tree level than the lowest at which a variable has appeared and that variable appears nowhere later in the search then it is ripe for elimination. The array LLVAR holding tree levels for variables is initialized in the first tree search to 1 for variables on the boundary of the root super-element and to a large integer for other variables and is kept up-to-date during the second search.

A third tree search is used to call ANAL1 (see sub-section 6.9) for each node that has more than two sons, or has a son that is a terminal node, or is a root with more than one son. ANAL1 works with a father node and its sons, having index lists for all of them available. It is called for a node having a terminal node for a son in case static condensation is suitable for this son, for our data structure does not permit eliminations at terminal nodes, so static condensation can take place

only if a node is an only son. Similarly we do not permit eliminations at root nodes, so these should always have just one son.

A fourth and final search checks that boundary lists of root super-elements contained through same interfaces are complete. During previous work within ANAL (and ANAL1) the whole list of global names at the interface will have been used and it is to be expected that static condensation will have taken place here to eliminate all internal variables. Therefore the boundary of the root super-element should contain all the variables in the list belonging to its father (or to itself if no static condensation has taken place). For robustness we therefore check this list and correct it if it is wrong. Note that ANAL calls ANAL1 for a root super-element as soon as it reached in the depth-first search, so that this check is made before the boundary list is used in the contained tree.

6.9 Subroutine ANAL1

Subroutine ANAL1 treats a part of the tree that consists of a node and its sons, assuming that lists of variables are available for all of them. It checks for any son having one or more variables not in the list of the father or any other son. Here static condensation is appropriate so a new node is created with the original son as its only son and the father as its father. Next ANAL1 considers all pairs of sons having one or more variables in common and chooses that pair which together have least variables to be sons of a new node having the original father as its father. The variable list for this new node is the union of the lists of its sons, less those not in the list of the father or any other son. The algorithm continues until the original father has a single son. Notice that it is

essentially a minimal degree algorithm because at each stage the freshly constructed element has minimal number of variables before any eliminations are performed.

We illustrate with the propellor blade of Figures 6 and 9, shown in Figure 12 with elements names shown circles and variable names shown circled

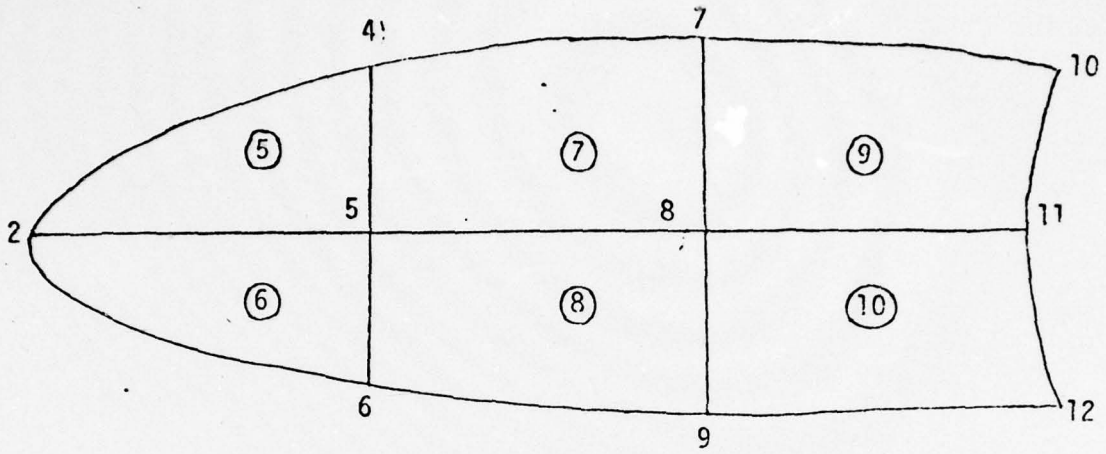


Figure 12 The elements and variables of the propellor blade.

and variable names shown without circles. The variables 10,11,12 are also included in the hub element and will be recognised as boundary variables because of appearing in the list associated with the father node. All the other variables appear in two or more elements so no static condensation occurs. The progress of the algorithm is summarized in Table 2. It begins with all pairs of adjacent elements, of which (5,6) contains the least variables, namely 2,4,5,6. Elements 5 and 6 are therefore combined into element 28 with variables 4,5,6 (since 2 can now be eliminated). This gives the new pairs of elements (28,7) and (28,8), each with 5 variables, in place of the first five pairs but leaves the remaining six pairs unaffected. We take (28,7) as the next pair for amalgamation into element 29 with variables 5,6,7,8 and new pairs (29,8),(29,9),(29,10)

replace all but the last three pairs. Continuing, we establish the tree shown in Figure 13.

Element pair	No. Vars	Element pair	No. variables	Element pair	No. variables	Element pair	No. variables	Element pair	No. variables	Element
5,6	4*	28								
5,7	5	28,7	5*	29						
5,8	6	28,8	5	29,8	5*	30				
6,7	6	—								
6,8	5	—								
7,8	6	•	•	—						
7,9	6	•	•	29,9	6	30,9	5*	31		
7,10	7	•	•	29,10	7	30,10	5	31,10	5*	32
8,9	7	•	•	•	•	—				
8,10	6	•	•	•	•	—				
9,10	6	•	•	•	•	•	•	—		

Table 2 Progress of the algorithm on Figure 12 problem.

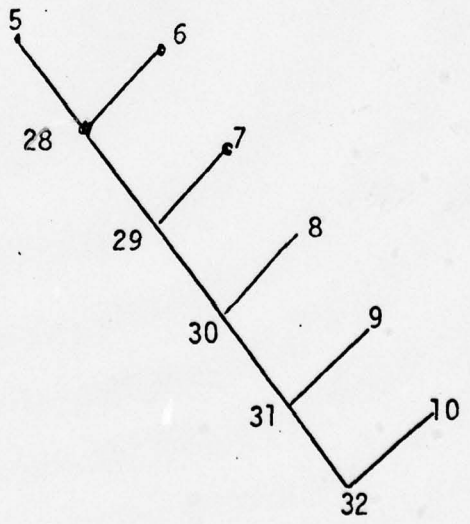


Figure 13 Tree established for Figure 12 problem

The code to implement this efficiently is quite long and we will not describe it here, except to remark that the array ISUP (equivalenced to array A) requires storage for

i) super-variables, each of which is a set of variables all of which belong to every element of some set of elements

ii) a list of pairs of adjacent elements.

Each entry in the first list requires two integers and each entry in the second list requires five integers. The storage requirement may therefore be great if there are a large number of elements (nodes). We envisage that the user will normally provide enough groupings of elements into super-elements for no node of the tree to have a very large number of sons, so do not expect this problem to be severe.

6.10 Subroutine SOLVE

Subroutine SOLVE is called directly by the user to solve one or more sets of equations associated with a root super-element that has already been successfully factorized, and the arguments have already been specified in section 3.

Forward substitution is performed under the control of the large elimination entries on the files which are associated with each tree (see explanation of data structure at end of section 4). As much as possible of the integer entry is read into array ELVAR and as much as possible of the real entry is read into array A, and the forward elimination continues until a same interface is reached, the arrays ELVAR and A being refilled whenever necessary. At a same interface the right-hand side vectors must be permuted so that forward substitution can continue using names local to the contained tree. This permutation is performed by subroutine CRSAME, described in sub-section 6.11, then elimination operations corresponding to the contained tree are performed in just the same way as those for the original tree, including the possibility of crossing another same interface. Whenever operations for a contained tree are completed, the same interface at its root must be crossed in the reverse direction, so

CRSAME is called for the inverse permutation and then operations in the containing tree can continue. Eventually operations for the original tree are completed and this means that forward elimination is finished.

If the root super-element for which SOLVE has been called has any boundary variables their values must be set after the forward substitution operations have been completed. This corresponds to their values being found by further forward substitutions in a containing tree and back-substitutions in that same tree. Their values cannot be input at the original call of SOLVE because then the forward substitution operations would corrupt them. This is a price that has to be paid for the facility of being able to call SOLVE for any root super-element and for being able later to embed any root super-element in larger structures through same interfaces.

Back-substitution is performed by reading the same file entries, but essentially in reverse direction and again using CRSAME to permute the vectors whenever a same interface is crossed.

6.11 Subroutine CRSAME

Subroutine CRSAME is called by SOLVE whenever forward or backward substitution involves crossing a same interface. It permutes a vector of reals under the control of pairs of integers (the local and global variable names). Its aim is to set all the numbers which are in B(FROM(I)), I=1,2,...,N on entry into B(TO(I)), while being reversible in the sense that a further call with the vectors TO and FROM reversed will restore the whole of B. We expect the length of B to be usually much greater than N and wish to perform only O(N) operations.

If K=TO(I)=FROM(J) for a pair of values I,J in the range [1,N] then it is necessary for the value of B(K) to be moved before it is overwritten. In fact it is appropriate to chain the moves. An example is given by the FORTRAN statements

```
B(10) = B(21)
B(21) = B(7)
B(7) = B(11)
```

for the case where 21,7 appear in both of arrays TO and FROM, 10 appears only in TO and 11 appears only in FROM. Without further action the original contents of B(10) would be lost and two copies of the original B(11) will be left. We therefore add extra statements to give the cyclic permutation

```
   T = B(10)
B(10) = B(21)
B(21) = B(7)
 B(7) = B(11)
B(11) = T
```

which will be reversed if TO and FROM are interchanged.

Simple Fortran of this kind illustrates the permutations but our actual code uses indirect addressing. The array VAR is assumed to contain all zeros and the code begins by setting VAR(TO(I))=FROM(I), I=1,2,...,N. A second loop negates these entries except where they correspond to the head of a chain. The final loop performs the actual permutation as a sequence of cycles, using indirect addressing. It also resets VAR to zero.

6.12 Input-output subroutines

Input and output is carried out by four subroutines IOGETR,IOGETI, IOPUTR,IOPUTI to get from file or put on file real or integer data respectively. Each has four arguments

ARY is an INTEGER (for IOGETI/IOPUTI) or REAL (for IOGETR/IOPUTR) array of length N used to transmit the data.

N is an INTEGER giving the number of reals or integers to be transmitted.

IFILE is an INTEGER indicating the file involved, having one of the values:

1. Main integer file
2. Integer work file.
3. Main real file
4. Real work file.

K is an INTEGER indicating the position within the file, addressing as if the file were a Fortran array, that corresponds to ARY(1).

These four subroutines presently merely transfer data to and from arrays in the common blocks CIF,CIF2,CRF,CRF2, respectively. The intention is that they be replaced by a virtual memory system that genuinely reads and writes to files out of main memory. The present subroutines will be quite effective on a computer system with a built-in paging system. Because of the way calls to these subroutines are organised, page thrashing is unlikely to occur. For instance during a call of SOLVE entries corresponding to each tree involved are together and are read once forwards and once (essentially) backwards.

7. Acknowledgements

This work would not have been undertaken without the initial interest expressed by P.S. Jensen and his continuing encouragement. I would like to express my thanks to him for this, and for his participation in the design and testing of the package and for his reading this document and suggesting two important additions. I would also like to thank the U.S. Air Force who have partially supported the work under contract F49620-76-C-0003; I.S. Duff who has read the draft of this report and suggested many improvement to the presentation; and the finite-element groups at Lockheed Palo Alto Laboratory and Boeing Computer Services, Seattle, with whom I have had very useful and stimulating conversations.

8. References

Irons, B.M. (1970). A frontal solution program for finite element analysis. Int. J. Numer. Meth. Engrg., 2, 5-32.

Ryder, B.G. (1974). The PFORT verifier. Software Practice and Experience, 4, 359-377.

A FORTRAN Virtual Storage Simulator for Non-Virtual Computers

Paul S. Jensen

Revised 4 March 1978

ABSTRACT

A software package called VMSYST for virtual type storage processing on a variety of computing systems is described. It utilizes a paging system for which the page and page buffer sizes can be conveniently adjusted to suit the application. For generality, the page buffer is held in a labeled common area. Except for the input/output routines, the package is written in standard FORTRAN for transportability. A gauche FORTRAN version of the input/output system is provided for simple testing but is not recommended for general use.

VMSYST was designed to support a sparse matrix package using a generalized frontal scheme. It proved very effective in that application but has also gained popularity in two other applications. It is currently operational on CDC 6000 series and Univac 1100 series computers.

1. INTRODUCTION

The concept of providing a high speed word addressable (core) memory of practically unlimited size has intrigued computer developers since the mid-1950's. Since economics has continuously forced rather severe limitations on the actual size of HSM (high speed memory), the concept of virtual memory evolved in which the unlimited size effect was achieved by judicious use of auxiliary storage devices. It was accomplished by hardware design in the late 1950's and appeared in the Burroughs B5500. It is also the standard design of the IBM 370 series computers.

Over the past decade, a great deal of experience in the operation characteristics of virtual memory systems has been gathered and there is a wealth of literature on the subject. (See [3, 6 or 14] for a general discussion.) Not all of the results have been desirable, in fact, in some scientific applications the results were almost disastrous [2, 5 and 12]. Other authors [7, 8, 9, 11, 13], however, provide glowing reports on virtual memory for scientific computation.

As a rule, it appears that if many small records are to be processed, a virtual system has some important advantages whereas for very large records, direct transfer to/from mass storage is more cost effective. Consequently, for scientific problems involving very large quantities of data it is often advantageous to have both, i.e., "virtual" mass storage files for parameters, indexes and tables, and "direct" files for large data blocks such as matrices.

In this paper we describe a system VMSYST of fairly simple FORTRAN subroutines which can conveniently be used to simulate virtual file operation. Unfortunately, the FORTRAN language does not have facilities for a number of specific, random access file manipulation processes that are needed and so certain support routines described in Section 6 must be provided for the use of VMSYST.

2. DATA BUFFERING SYSTEM

2.1 Page Buffer

The system presented here utilizes a page buffer array in a labeled common block for the actual data (in pages) held in the HSM (high speed memory). It is likely that this buffer would more appropriately be maintained in an extended or supplemental core storage on computing systems offering such a feature. Even high speed drum storage could conceivably be used for this purpose, particularly if a direct data path between the drum and mass storage is available.

For security and control, the contents of the buffer can only be modified by the virtual system routines. Thus, when an application program requests data resident in mass storage, it is first moved to the buffer and then to the array provided by the application program. This double movement of data is the major price paid for the benefits of the virtual system.

All data in the virtual system is partitioned into pages, which are blocks of consecutive data words of a fixed page size. Pages residing in the buffer are called active pages. Inactive pages are resident in auxiliary storage only.

2.2 Page Table

The buffer storage is partitioned into blocks, each being the size of one page. Corresponding to each of these blocks is one column of control information (see Table 2.1) in a page table residing in a distinct labeled common block. The page table is the key mechanism for keeping track of what data is in the buffer and its status.

3. STORAGE CONTROL SYSTEM

An application issues the I/O requests which are either file management or data transfer requests. Data transfer requests are used to move

TABLE 2.1

Control Information Maintained in Column j
of the Page Table for an Active Page of Class c

ITEM	PURPOSE
1	Next older active page of class c (or o)
2	Youngest active page of class j (or o)
3	Virtual mass storage location of this active page (64·Virtual address + File No.)
4	Physical location of this active page in mass storage (physical units, e.g., sectors)
5	Next younger active page of class c (or c + total no. of active pages)
6	Next older active page (or o)
7	Next younger active page (or o)

data between a specified array and mass storage at a specified virtual location. When a data transfer request is issued, the following steps define the fundamental transfer process:

1. Determine if the page corresponding to the virtual location is active. If it is, denote it by P and then go to step 7

2. Look for an empty active page. If one is found, denote it by P and then go to step 5

3. Find the "oldest" active page P. If it does not differ from its corresponding inactive copy, then go to step 5

4. Copy P to its virtual location in mass storage (bump the page)

5. Enter the control information of the page corresponding to the virtual location of the data to be transferred in the page table column for page P.

6. If the data transfer is input from mass storage or the transfer data does not include all of page P, and if there is an inactive copy of the page P in mass storage, then transfer the inactive copy to the buffer, thereby making P fully active.

7. If the data transfer is input from mass storage, then copy that portion of the transfer data contained in P to the application program array.

8. If the data transfer is output to mass storage, then copy that portion of the transfer data for page P from the application program array to P.

9. If not all of the transfer data has been copied, then update the virtual location and size to reflect the remaining transfer data and go to step 1

10. Job complete.

It should be noted that step 3 defines the so-called "paging algorithm" discussed widely in the literature. It is the criteria used to determine which page (or pages) is to be"bumped". Here we use the simple criterion of "age". Substantially more complicated criteria have been devised [1] with varying success, depending upon the application.

The search of the page table required in step 1 has also received considerable attention [10]. In VMSYST, an approach similar to hashing without the possibility of conflicts is used. Each page p is assigned to an equivalence class $c_p = p \bmod m + 1$, where m is the page capacity of the page buffer. In this way, only the active pages of class c_p need to be checked in step 1 for any page p. A bidirectional linked list is maintained for the active pages of each class to facilitate the search.

When a large block of data is to be transferred, it is possible for the fundamental process described above to bump pages that will subsequently be needed for the transfer. This eventuality is overcome by paritioning the data block according to page destination and first transferring those partitions corresponding to active pages.

This innovation always has a beneficial effect on the actual input/output volume to mass storage, however, at the cost of considerably more page table scanning. Statistical evidence has not yet been gathered to indicate the overall cost impact.

4. FILE MANAGEMENT

The processes of establishing communication with auxiliary storage devices, cataloging them for permanent retention of data, locating specific data on previously cataloged files, etc., are very machine dependent. Consequently, machine independent routines for such file management operations cannot be generated. However, a file access table which can provide sufficient descriptive data for a great variety of auxiliary storage equipment can be devised and used by "transportable" FORTRAN computer code.

The information retained for each file used in this virtual memory system is outlined in Table 4.1. It takes into account the fact that data in auxiliary storage is normally in blocks of standard computer words (SCW's) which are called physical block units (PBU's) here. The SCW is taken to be the standard addressing unit of high speed memory in a computing machine, e.g., 32 bits on IBM computers, 60 bits on many CDC computers and 36 bits on Univac computers.

The file identifier is also a machine dependent form which is often an alphanumeric. The sixth and seventh words in the information list are currently provided for the identifier with the eighth unassigned, making an expansion to three words simple if needed for some computing systems. For convenience, each file is referenced by its column number in the file table rather than its identifier. The last item is used as an access key to help prevent accidental data loss. Presently it is initially set to one of "get", "put" or "both".

TABLE 4.1

File Information Maintained in the File Access Table for Each File (Auxiliary Storage Unit) Defined

PBU - Physical Block Units (Sectors)
SCW - Standard Computer Word

ITEM	PURPOSE	UNITS
1	No. of PBU's per page	
2	Current position	PBU
3	Next free position	PBU
4	Capacity	SCW
5	PBU (Physical block unit) size	SCW
6, 7	File Identifier (zero if inactive)	
8	Not presently used	
9	Access key ("get", "put", or "both")	

5. FORTRAN ROUTINES

When the virtual memory system is operational on a computer system, the following fourteen processors are all that an application program will have occasion to reference:

```
IOSET
IOSTAT
IOOPEN(FILEID,FILENO,FILINF)
IOCLOS(FILENO)
IOPUTx(A,NA,FILENO,VLOC)
IOGETx(A,NA,FILENO,VLOC)
IOCLRx(A,NA,FILENO,VLOC)
IOADDx(A,NA,FILENO,VLOC)
IOSCLx(A,NA,FILENO,VLOC)
```

where x is either I or R depending upon the type of data in A. The first eight routines pertain to file handling and the movement of data and the last six provide special numerical operations on data in virtual memory.

5.1 Initialization

Initial values for the tables and parameters of the system are established by one call of the form

```
CALL IOSET.
```

This subroutine sets the parameter values as indicated in Table 5.1 and clears the FILE, PAGE and BUFFER tables.

Auxiliary storage files are opened by a call of the form

```
CALL IOOPEN(FILEID,FILENO,FILINF),
```

where

FILEID is a two word alphanumeric file name supplied by the application program,

FILENO is a file reference integer for the file supplied by IOOPEN,

FILINF is a five word integer file description array, see Table 5.2

IOOPEN calls upon a special routine DMDAST (See Section 6) for establishing an auxiliary file via executive requests. If a routine of this nature is not available at a particular installation, a suitable modification of IOOPEN will be required.

5.2 Data Transfer

The fundamental data transfer operations are GET data from virtual memory and PUT data into virtual memory. Data transfer is accomplished by calls of the form

```
CALL IOGET x (A,N,VF,VL)
```

and

```
CALL IOPUT x (A,N,VF,VL),
```

where

x is "R" or "I" for real or integer data transfer,

A is an array of length N and type corresponding to x above,

VF is an integer file number established by IOOPEN and

VL is a virtual location (in standard computer words) on VF.

TABLE 5.1

VMSYST Key Parameters (Integer)

NAME	BASE VALUE	PURPOSE
NFIL	8	Maximum number of simultaneously opened files (no. columns in file table array FILET)
NBUFR	22400	Size of buffer array
PAGESZ	896	Size of each page (common multiple of auxiliary file sector sizes, e.g., Univac-28, CDC-64)
INPRE	1	No. of integers per real (standard machine word
NBPIN	35	No. of bits available for positive integers ($\log_2$ IMAX, where IMAX is the largest machine representable integer)

TABLE 5.2

File Descriptor Array FILINF (see also Sec. 6.3)

ITEM	PURPOSE	SPECIFICATION
1	Equipment type	-1 Tape 0 Disc 3 Extended Core
2	Permanency option	0 Temporary 3 Existing 6 New Permanent
3	Capacity	In standard machine words
4	Tape reel ID	If appropriate
5	Access	"GET" Read only "PUT" Write only "BOTH" General

The actual movement of data is handled in units of standard computer words. Thus if A is an integer array in a system having integer words shorter than standard, then

$$INPRE\cdot(1+[(N-1)/INPRE])$$

integers will actually be transferred, where INPRE is the number of integers per real (see Table 5.1) and [x] is the largest integer not exceeding x.

The transfer of data between auxiliary storage and the buffer is handled by special "position", "read" and "write" utilities DMPAST, DMRAST and DMWAST described in Section 6.

5.3 System Status

It is helpful in large processes to obtain a brief summary of the IO activity from time to time. VMSYST maintains a number of statistics in this regard which are displayed by a call of the form

```
CALL IOSTAT.
```

5.4 File Release

When operations involving a specific file are finished, the file may be released from the system by a call of the form

```
CALL IOCLOS(FILNO)
```

Where the absolute value of FILNO is the reference number established by IOOPEN.

All of the active pages corresponding to the file that may differ from the inactive ones are transferred to the file. If FILNO is positive, then the entry in the file table FILET for the file is deleted, and the file is also released (made inactive if permanent or purged if temporary) from the executive operating system by means of special support routine DMFAST (see Section 6).

5.5 Special Functions

Three special arithmetic processes are included for numerical applications. The inclusion of these processes in VMSYST had almost no impact on the implementation, but are of tremendous value to numerical application programs.

To clear an area of virtual memory (i.e., to set the contents to one prescribed value), a call of the form

CALL IOCLR x (A, N, VF, VL)

is used where scalar A is the prescribed value and the other arguments are as described in Section 5.2.

A scaled vector $s \cdot A$ may be added to the contents of an area in virtual memory by a call of the form

CALL IOADD x (A, N, VF, VL),

where the scalar s is held in $A(N+1)$.

Finally, the contents of an area in virtual memory may be multiplied by a scalar constant A by a call of the form

CALL IOSCL x (A, N, VF, VL) .

6. AUXILIARY STORAGE INTERFACE

Unfortunately, there are no standard facilities in the FORTRAN language for dealing with general auxiliary storage media. Consequently, a set of FORTRAN subroutines called DMGASP [4] with facilities for the necessary communication with the executive operating system has been constructed at the Lockheed Research Laboratory to fill this gap. Several of these are used in support of the virtual memory system VMSYST discussed here.

In this section we describe the functions performed by these support routines so that approximate duplicates may be constructed elsewhere for support of VMSYST. This description is intended to be minimally sufficient and interested readers are urged to refer to [4] for a comprehensive description.

For this discussion, we shall frequently refer to an "auxiliary storage device" which we abbreviate to ASD. We specifically exclude the normal user input (card reader, teletype, etc.) and output (printer, plotter, etc.) as ASD's. In the broad sense, of course, these are also ASD's, but their management is best relegated to the operating system.

6.1 Fundamental Operations

DMGASP performs the following five fundamental auxiliary storage operations, where we include alternative nomenclature for the operations in parentheses:

1. Declare (assign, attach, activate, open) an ASD,
2. Free (release, deactivate, close) an ASD,
3. Write (store, put) data on an ASD,
4. Read (copy, get) data from an ASD and
5. Position an ASD.

The write and read operations always transmit data directly between an ASD and a storage block in HSM. Thus the user program is not burdened with considerations of hidden buffering that often impedes high volume I-O.

6.2 Device Reference System

In operations 2-5 above, the ASD is designated by its LDI (logical device index), which is an integer in the range 1, 2, ... , MAXLDI*. If an ASD is also given an (external) alphanumeric name, the connection (equivalence) of that and the LDI is established in the declaration operation.

The declaration is always the first operation performed on an ASD. When an ASD is declared, its attributes (see below) are entered in an auxiliary storage table (AST) to facilitate all succeeding operations. They remain there until a free operation is performed. After an ASD is thus deactivated, its LDI may be used in another declaration with another ASD.

6.3 Storage Device Attributes

The four attributes:

TYPEX	ASD type index (-1 to 3)
OPTX	ASD option index (0 to 12)
LIMIT	ASD capacity (in words) and
REEL	Tape reel identifier

are maintained for each declared ASD. The typical default condition is to set all four attributes to 0. That yields a temporary, sector addressable mass storage device with a system determined default capacity. If a tape is declared (TYPEX = -1), then REEL = 0 implies the use of a new blank tape. A previous reserved tape is declared (mount request) by setting REEL to the literal (alphanumeric) tape identifier, e.g., REEL = "1 2 3 4 5 6". The interpretation of the remaining type and option indexes are provided in Tables 6.1 and 6.2

---

* MAXLDI is an internal parameter set at the installation. It is generally larger than 32.

TABLE 6.1

Type Index for Auxiliary Storage Devices

Index (TYPEX)	Type of Equipment
-1, -2	Magnetic tape (7/9 track)
0	Sector-addressable, high-capacity mass storage
1	Sector-addressable, high-speed mass storage
2	Word-addressable mass storage
3	Extended core storage

TABLE 6.2

Option Index for Auxiliary Storage Devices (ASD's)

Option (OPTX)	Suitable Type	Interpretation
0	All	Temporary ASD
1	-1, -2	Write enable reserved tape
2	All	Identify previously assigned ASD
3	0	Access previously cataloged ASD
4	0	Exclusive access to previously cataloged ASD
5	0, -1, -2	Permanent tape or disc file (private)
6	0	Permanent disc file (public)
7	0	Permanent disc file (private, read only)
8	0	Permanent disc file (public, read only)
9-12	0	Like 5 - 8 but permanency conditional upon job completion

6.4 Operation Naming Convention

The six ASD operations performed by DMGASP are indicated by the first letters in their names as indicated in Table 6.3.

TABLE 6.3

Auxiliary Storage Operation Designators

Designator	Operation
D	Declare (assign, attach, activate, open)
F	Free (release, deactivate)
P	Position
W	Write (store, put)
R	Read (copy, get)
E	End file (tape)
L	List DMGASP operational data

The operations are carried out by subroutine calls to entry points in DMGASP having the form

```
CALL DMxAST(L, M, N)
```

where x is any of the letters DFPWREL. There are always three arguments which serve operation dependent functions. Table 6.4 lists the defined functions of the arguments.

TABLE 6.4

Functions Served by the Three Arguments
L, M, and N of a DMGASP Operation
DMxAST(L, M, N)

(Operation designator x is one of DFPWREL, see Table 6.3)

Argument	Operation Designator	Function
L	L	LOSD-List Operation Status Descriptors if non-zero
	All Others	LDI-Logical Device Index
M	D	EDNAME-External Device Name (Optional)
	F	DELETE-0:release, 1:decatalog
	P	LOC-Location (Disc), Tape File Number (TFN), or 4096(TFN-1)+Record-1
	W, R	ARRAY-HSM (Core) Data Block
	E	Ignored (dummy)
	L	LPKT-List(LPKT+1) words of the file information table
N	D	DDPARS-4 word attribute table
	F, E	Ignored (dummy)
	P	MODE-0 sectors from start 1 words from start -1 words from current pos. (Disc)
	W, R	SIZE-Array Size
	L	LTAB-List Auxiliary Storage Table if non-zero

6.5 Supplemental Operations

In addition to the operations discussed in Section 6.4, there are some supplemental operations related to those of DMGASP. Notable among these is accessed by

```
LSECT = LMSECT (LDI)
```

which provides the sector size of the declared disc file with logical device index LDI.

6.6 Operations Used in VMSYST

The operations DFPWR (Table 6.3) and the supplemental operation (Section 6.5) are referenced by VMSYST. These references are confined to four VMSYST subroutines as indicated in Table 6.5 Adaptations of VMSYST to computing systems not having DMGASP may be made either by simulation of the operations of Table 6.5 or by suitable modification of the VMSYST subroutines listed in Table 6.5

TABLE 6.5

DMGASP References in VMSYST

Subroutine	DMGASP Operations
IOOPEN	D,P,LMSECT
IOCLOS	F,P,W
IOPAGE	R,P,W

7. ACKNOWLEDGMENTS

The author expresses his thanks to Dr. John K. Reid for several helpful suggestions used in the design of VMSYST. In particular, Dr. Reid's suggestions for page table searching were most helpful.

References

1. Belady, L. A., "A Study of Replacement Algorithms for a Virtual Storage Computer," IBM Syst. Jnl. 5, 2 (1966)

2. Coffman, E. G. and L. C. Varian, "Further Experimental Data on the Behavior of Programs in a Paging Environment," Comm. ACM 11, 7 (1968) 471-474

3. Denning, P. J., "Virtual Memory," (ACM) Computing Surveys 2, 3 (1970) 153-189

4. Felippa, C. A., "The Input-Output Manager of the Nostra Data Management System, Univac 1100-Series Version Reference Manual," Report LMSC-D556430, Lockheed Palo Alto Research Lab (1977)

5. Fine, G. H., et al., "Dynamic Program Behavior under Paging," Proc. ACM 21st Nat. Conf., Thompson Book Co., Washington, D.C. (1966) 223-228

6. Goldberg, R. P., "Virtual Machine Systems," Report MS-2687, MIT Lincoln Laboratory (Sept 1969)

7. Hatfield, D. J. and J. Gerald, "Program Restructuring for Virtual Memory," IBM Systs. Jnl. 10 (1971) 168

8. Kortzeborn, Robert N., "Virtual Computing and Its Importance to Scientists," Report 320-3308, Palo Alto Scientific Center, IBM Data Processing Division (Nov 1972) 127 p

9. Kortzeborn, R. N. and P. J. Friedl, "The Advantages of Using the IBM System/360 Computers for Large Scale Scientific Problems," Report 320-3233, Palo Alto Scientific Center, IBM Data Processing Div. (Oct 1967)

10. Maruyama, K. and S. E. Smith, "Analysis of Design Alternatives for Virtual Memory Indexes," Com.of the ACM, 20, 4 (1977) 245-253

11. McGrath, M., "Virtual Machine Computing in an Engineering Environment," IBM Systs. Jnl. 11 (1972) 131

12. McKellar, A. C. and E. G. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems," Comm. ACM 12, 3 (1969) 153-165

13. Parmelee, R., "Virtual Machines: Some Unexpected Applications," Proc. of the 1971 IEEE Int. Comp. Soc. Conf., Boston, Mass.

14. Parmelee, R. P., T. I. Peterson, C. C. Tillman and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts," IBM Systs. Jnl. 11 (1972) 99

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR- 78-1149 ✓	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): SPARSE SYMMETRIC MATRIX PROCESSING	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER: LMSC-D626184 ✓
7. AUTHOR(s): Paul S. Jensen and John K. Reid	8. CONTRACT OR GRANT NUMBER(s): F49620-76-C-0003 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Lockheed Palo Alto Research Laboratory ✓ 3251 Hanover Street Palo Alto, California 94304	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A3
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332	12. REPORT DATE: May 1978
	13. NUMBER OF PAGES: 95
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Sparse matrices, wave front, factorization, virtual, camparison, testing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document is extensive report describing a new sparse matrix processing system for very large symmetric matrices. It includes a description of a virtual memory system used in support of the system and results from comparison tests with a profile matrix processor on problems arising from finite-element analyses of structures.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

CLASSIFICATION OF THIS PAGE (When Data Entered)

END